UNIVERSITY OF CALIFORNIA

Los Angeles

On the Correctness of Transactional Memory Algorithms

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Mohsen Lesani

2014

ABSTRACT OF THE DISSERTATION

On the Correctness of Transactional Memory Algorithms

by

Mohsen Lesani
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2014
Professor Jens Palsberg, Chair

Transactional Memory (TM) provides programmers with a high-level and composable concurrency control abstraction. The correct execution of client programs using TM is directly dependent on the correctness of the TM algorithms. In return for the simpler programming model, designing a correct TM algorithm is an art. This dissertation presents techniques to prove the correctness or incorrectness of TM algorithms. In particular, it contributes to the specification, safety criterion, testing and verification of TM algorithms.

We introduce a language for architecture-independent specification of synchronization algorithms. An algorithm specification captures two abstract properties of the algorithm namely the type of the used synchronization objects and the pairs of method calls that should preserve their program order in the relaxed execution.

We introduce the markability correctness condition as the conjunction of intuitive invariants: write-observation and read-preservation. We prove the equivalence of markability and opacity correctness conditions. Decomposition of the correctness condition supports modular and scalable verification.

We identify two pitfalls that lead to violation of opacity: the write-skew and write-

exposure anomalies. We present a tool called Samand that automatically finds traces of such bug patterns. Using Samand, we show that the DSTM and McRT algorithms suffer from the write-skew and write-exposure anomalies respectively.

We present a sound program logic called synchronization object logic (SOL) that supports reasoning about the execution order and linearization orders of method calls. It provides inference rules that axiomatize the properties and the interdependence of these orders and also the properties of common synchronization object types. We present the derivation of markability in SOL as a sound syntactic proof technique for opacity. We use SOL to prove the markability and hence opacity of the TL2 algorithm in PVS.

The dissertation of Mohsen Lesani is approved.

Todd Millstein

Glenn Reinman

Hans Boehm

Edward Effros

Jens Palsberg, Committee Chair

University of California, Los Angeles

2014

To Niloufar, Shahnaz and Mehdi

Contents

List of Figures

List of Theorems

Publications during the PhD studies:

- Automatic Atomicity Verification for Clients of Concurrent Data Structures

  Mohsen Lesani, Todd Millstein, Jens Palsberg

  CAV'14 (Computer Aided Verification Conference 2014)

- Semantics-preserving Sharing Actors

  Mohsen Lesani, Antonio Lain

  AGERE'13 (Actors, Agents, and Decentralized Control Workshop 2013)

- Specifying Transactional Memories with Nontransactional Operations

  Mohsen Lesani, Victor Luchangco, Mark Moir

  WTTM'13 (Theory of Transactional Memory Workshop 2013)

- Write-observation and Read-preservation TM Correctness Invariants

  Mohsen Lesani, Jens Palsberg

  WTTM'13 (Theory of Transactional Memory Workshop 2013)

- MrCrypt: Static Analysis for Secure Cloud Computations

  Sai Deep Tetali, Mohsen Lesani, Rupak Majumdar, Todd Millstein

  OOPSLA'13 (Object-Oriented Programming, Systems, Languages & Applications Conference 2013)

- Proving Non-opacity

  Mohsen Lesani, Jens Palsberg

  DISC'13 (DIStributed Computing Conference 2013 - LNCS 8205, Transact'13 (Transactional Computing 2013)

- A Framework for Formally Verifying Software Transactional Memory Algorithms

  Mohsen Lesani, Victor Luchangco, Mark Moir

  CONCUR'12 (Concurrency Theory Conference 2013)

- Putting Opacity in its Place

  Mohsen Lesani, Victor Luchangco and Mark Moir

  WTTM'12 (Theory of Transactional Memory Workshop 2012)

- Communicating Memory Transactions

  Mohsen Lesani, Jens Palsberg

  PPoPP11 (Principles and Practice of Parallel Programming Conference 2011)

BIOGRAPHICAL SKETCH

Mohsen Lesani obtained his Mathematics and Physics diploma from the National Organization for Development of Exceptional Talents of Iran in 1996. He obtained his bachelor degree in Software Engineering from University of Tehran in 2004 and his masters degree in Artificial Intelligence from Sharif University of Technology in 2006. He has teaching experience as a lecturer at University of Kerman. He has research experience at EPFL, HP labs and Oracle (Sun) Labs. His PhD research was supported by UCLA and IBM Research. He was awarded the UCLA computer science department Outstanding Graduate Research Award in 2014.

Chapter 1

Introduction

Transactional Memory (TM) [41, 71] provides programmers with a high level concurrency control abstraction. Programmers can simply declare certain blocks of code as transactions and the TM runtime guarantees that transactions execute in isolation. The use of TM provides atomicity, deadlock freedom, and composability [35], and increases programmer productivity compared to use of locks [65, 68]. Researchers have developed formal semantics [1, 58, 47] and a wide variety of implementations of the TM interface in software [39, 38, 18, 69, 19], hardware [32, 3], and software/hardware hybrids [5, 56, 15]. Recently, industry is adopting TM. IBM supports TM in its Blue Gene/Q processor [33], and Intel supports transactional synchronization primitives in its new processor microarchitecture Haswell [14]. The C++ transactional memory study group (SG5) is introducing transactional constructs to the C++ language.

The TM runtime takes the responsibility of managing the consistency of the shared state. Therefore, the correct execution of client programs using the TM interface is dependent on the correctness of TM algorithms. In return for the simpler programming model, designing a correct TM algorithm is an art. Algorithm designers employ different techniques to provide the TM interface efficiently. They interleaves transactions as much as possible, while guaranteeing non-interleaving semantics. Thus, subtle but fast algorithms are favored over

simpler ones. The subtlety of the algorithms makes them prone to intricate bugs. Thus, the correctness of TM algorithms is both a central and a formidable problem. This dissertation presents techniques to prove the correctness or incorrectness of TM algorithms. In particular, it contributes to the specification, safety condition, testing and verification of TM algorithms.

Specification. Precise specification of algorithms is the first step towards testing and verification of them. The current literature on synchronization and particularly TM algorithms usually presents algorithms in prose or architecture-dependent code. The effect can be unfortunate: under- and over-specification of the algorithm and therefore misunderstanding, irreproducibility and unportability.

We introduce a language for architecture-independent specification of synchronization algorithms. An algorithm specification captures two abstract properties of the algorithm namely the type of the used synchronization objects and the required program orders. The language supports the following common synchronization object types: basic register, atomic register, atomic cas register, lock, try-lock, strong counter, basic set and basic map. Each method definition involves the declaration of the pairs of method calls whose order in the program should be preserved in the relaxed execution. The specifications can be studied, and verified once and for all, independently of the implementation of the synchronization objects and the memory models of the compiler and the architecture. Compilers can optimize implementations of the synchronization object types and also the number, type and position of memory fences. We specify several well-known TM algorithms in the language.

We define the semantics of specifications as a set of execution histories. We define the denotational semantics of a specification as a set of constraints that enforce both the structure of the the program and the safety of the used objects. The semantics is compositional, models true concurrency and allows relaxed execution.

Safety. A TM algorithm should guarantee that every concurrent execution of an arbitrary set of client transactions is indistinguishable from a sequential execution of them.

Safety conditions for TM such as opacity [28], VWC [44], TMS1 and TMS2 [22] define the indistinguishably criterion and the set of correct histories. Lesani et al. [51] proved that opacity is stronger than TMS1 and weaker than TMS2. Verification of TM algorithms is a formidable problem in part because the target correctness criterion is a monolithic complicated condition. Is there an intuitive decomposition of TM correctness conditions? What are the separate invariants that the TM designers should maintain? Decomposition of the correctness condition informs designers by showcasing different aspects of correctness and helps them concentrate on maintaining one aspect at a time. In addition, separation has obvious benefits of modularity and scalability for verification. In an early work, Tasiran [74] presented a decomposition of the correctness condition for a specific class of algorithms.

We present intuitive invariants for the correctness of TM algorithms. We say that a history is markable if there is a specific ordering relation called marking such that certain invariants are satisfied. We prove the equivalence of markability and opacity. At a high level, the first invariant called write-observation requires that each read operation returns the most current value and the second invariant called read-preservation requires that the location which is read is not overwritten in a certain interval.

Testing. Algorithm design is an iterative process of trying alternatives, fixing issues and improving the performance. A testing tool can assist algorithm designers during both the design and the maintenance of the algorithm. Manovit et al. [53] and Lourenco et al. [52] applied random testing to TM algorithms.

We identify the write-skew and write-exposure anomalies as two pitfalls that lead to non-opacity. We present a tool called Samand that automatically finds traces of such bug patterns. The tool inputs a TM algorithm, a program and a test assertion. The test assertion can be a partial correctness condition such as negation of a bug pattern. If there is an execution of the test program that violates the test assertion, Samand outputs the trace of a violating execution. Samand translates concurrent execution to constraints and employs Z3 SMT solver [17] to solve the constraints. Using Samand, we show that DSTM [39] suffers

from the write-skew anomaly and McRT [69] suffers from the write-exposure anomaly. These results may be surprising because previous work [31, 30, 24] considered abstract version of DSTM and McRT and proved their correctness.

Verification. Verification of TM algorithms has been a topic of recent attention. Researchers have employed model checking, automatic invariant generation and theorem proving to verify the correctness of TM algorithms. Model checkers from Cohen et al. [11, 12], and Guerraoui et al. [29, 31, 30] were the pioneering approach to verification of TM. Subsequently, the same approach was taken by O'Leary et al. [62] and Baek et al. [4]. Model checking can automate the verification process but is either based on assumed properties about the TM algorithm or only scalable to a finite number of threads and locations or simplified algorithms. Later, Emmi et al. [24] tried to automatically infer invariants that are strong enough to entail the correctness criterion. Compliance of the algorithms with the specification can be easily checked if the proper invariants can be automatically generated. On the other hand, this work reported resorting to simplified algorithms due to scalability issues. Later, Lesani et al. [50] presented a machine checked theorem proving framework based on I/O-automata and proved the correctness of NORec TM algorithm [16]. The framework can be employed to verify realistic algorithms but requires translation of the algorithm to a transition system and more importantly, the process involves coming up with non-trivial invariants.

Aside from modeling simplified algorithms and limited scalability, all previous works on verification of TM algorithms work with semantic models. We believe that studying TM algorithms can benefit from focusing on the syntactic specification of an algorithm, syntactic description of correct algorithm, and a deduction mechanism to reason about algorithm correctness using the structure of the algorithm and the properties of used synchronization objects.

As we review in the related works chapter, the previous works on concurrent program logics support several forms of local reasoning but do not support assertions for the execu-

tion order or the linearization order of method calls across threads. These assertions are particularly essential for reasoning about TM algorithms. A concurrent execution of a set of transactions is correct if there is an indistinguishable sequential order of the transactions. The sequential order is determined by the execution order or the linearization order of certain method calls in the transactions. We present a program logic called synchronization object logic (SOL) that supports reasoning about the execution overlap, execution order and linearization orders of method calls. It provides inference rules that axiomatize the properties and the interdependence of these orders and also axiomatize the properties of common synchronization object types. We prove the soundness of the logic. SOL derives valid conclusions from valid premises.

We define the markability assertions in SOL and prove that a TM algorithm is opaque if markability can be derived for its specification. Therefore, deriving the markability is a sound syntactic proof technique for opacity of TM algorithm specifications.

We formalize SOL in PVS [64] and use it to machine-check the markability of the well-known TM algorithm TL2 [18]. SOL is applicable beyond TM, particularly to algorithms for mutual exclusion. As evidence, we prove the mutual exclusion property of the Dekker algorithm.

In the following chapters, we first define the specification language. Then, we consider safety conditions and introduce markability. Then, we present our testing approach based on bug patterns. Finally, we introduce our logic, present the markability proof technique and use the logic to prove the markability of TM algorithms.

Chapter 2

Synchronization Object Language

2.1 Introduction

Precise specification of algorithms is an important prerequisite to understanding, testing and verification of them. The current literature on synchronization algorithms and particularly TM algorithms usually presents algorithms in architecture-dependent code or prose. The effect can be unfortunate: *under- and over-specification* of the algorithm and therefore misunderstanding, irreproducibility and unportability. We present some instances at the end of this subsection. Separation of specification and implementation is a classical design principle. In this chapter, we introduce a language for architecture-independent specification of synchronization algorithms. The language supports the common *synchronization object types*, and allows *relaxed execution*. We present the syntax and then the semantic of the language.

Syntax. We introduce a language for the specification of synchronization algorithms. A specification is comprised of three sections: the typing section, the definitions section, and the program section. The typing section is the type declarations of the synchronization objects that the algorithm uses. The definitions section is the definitions of the methods of the algorithm. Each method definition involves the declaration of pair of method calls whose

order in the program should be preserved in the relaxed execution. The program section defines the parallel program or programs that call the defined methods.

In particular, the specification captures two abstract properties of the algorithm namely the type of the base objects and the required orders. A synchronization object type such as atomic register declares the safety and liveness properties that the objects of the type guarantee. Our language supports the following synchronization object types: basic register, atomic register, atomic cas register, lock, try-lock, strong counter, basic set and basic map. The type and its properties are abstract from the implementations of the type. Similarly, the required program order is abstract from the memory model or the fences needed to satisfy it. The specifications can be studied, and verified once and for all, independently of the implementation of the base objects and the memory models of the compiler and the architecture. Compilers can benefit from optimized implementations of synchronization object types and optimize the memory layout of the base objects. They can also optimize the number, type and position of fences to satisfy the program order.

A TM algorithm specification should define the four methods of the TM interface: init, read, write and commit that initialize the transaction, read from a location, write to a location and attempt to commit the transaction. We consider an arbitrary set of transactions as the client parallel program. We specify several well-known TM algorithms in the language.

Semantics. We adopt the notion of execution history [40, 28] as the representation of a concurrent execution. First, we remind execution histories and their operations and relations.

Then, we define synchronization object types. We present the semantics of linearizable and basic objects as sets of execution histories. We define abstract object types and concrete synchronization objects. We present the interface and the sequential specification of each abstract object type. For each synchronization object type, we present lemmas that characterize the properties of its execution histories.

Based on the above definitions, we define the semantics of specifications as a set of

execution histories. We define the denotational semantics of a specification as a set of constraints that enforce both the structure of the the program and the guarantees of the base objects. The semantics is compositional i.e. it is abstract from and can be modularly augmented with new object types. It models true concurrency i.e. it considers a pair of invocation and response events for each method call. It allows relaxed execution i.e. it supports out-of-order execution of method calls that are not specified to be ordered.

TM Algorithm Specification Pitfalls. Now, we present some instance of pitfalls in the specification of TM algorithms that we have encountered in the literature.

The algorithms that are tailored for specific architectures and memory models [18, 69, 19] can under- and over-specify the algorithm. Some orders of execution that are implicitly provided by one memory model may need explicit fences in other memory models. Thus, some important orders may be left unspecified. For example a subtle required order is unspecified in TL2 [18] as noted later [31]. Therefore, porting algorithms can introduce bugs. For example, an earlier release of the STAMP benchmarks [55] had an incorrect port of the TL2 algorithm from SPARC to x86. Similarly, some orders that needed fences in one memory model may be implicitly provided by another. Extra fences hinder performance.

Striving for efficiency, several objects are packed to or share the same memory location. To avoid false sharing phenomenon, objects are explicitly padded. The details of object layout can obfuscate the algorithm intents. As observed by previous work, in the original TL2 paper,"the authors maintain the version number and the lock bit of every variable in the same memory word" [31], thus, the order of checking the lock and the version of read locations is ambiguous in the commit procedure. In our specification, we treat the lock and the version as separate registers and make the orders explicit.

Imprecise specifications lead to irreproducibility. TL2 algorithm [18] and DSTM algorithm [39] are explained in prose. Therefore, rewritten specifications of it in the literature are inaccurate or incorrect. For example, there is no visible reads in the DSTM algorithm but the specifications in [31] and [24] abort the visible readers during the validate command.

In [30], DSTM is specified with no dynamic object allocation while the original algorithm [39] is fundamentally based on the indirection that is obtained from dynamic creation of locator objects. In addition, there is no distinction between read and write operations in the specification. The read operation simply calls the write operation, thus a read acquires the location similar to a write. This is while readers do not acquire the location in the original algorithm. In the specification, the commit operation writes to every location that is written to during the transaction. This is while commitment is done by a single compare-and-swap in the original algorithm. As another example, TL2 algorithm [18] is based on version numbers while the specifications of TL2 in [31] and [24] replace the version number concept with the unprecedented notion of modified sets. Furthermore, there has been a typo of writing *os* instead of *ls* in the TL2 transition system in [31]. The follow up work [24] that rewrites this specification, incorrectly fixed *os* to *ws* and thus verified a different algorithm. In [30], the check that the version of the read location is less than the read version is replaced with an equality check. This restricts the concurrency of the algorithm. A local array *lver* is introduced that is written during the read operations and checked during the commit procedure. This local array does not exist in the original algorithm.

2.2 Syntax

Now, we introduce the syntax of the language. We define the structure of a specification and then define a TM algorithm specification as a specific specification. To have more concise specifications, we extend the syntax with syntactic sugar. We present the specifications of Dekker mutual exclusion algorithm and TL2, TL2 variant, DSTM, DSTM (visible reads), and McRT TM algorithms.

2.2.1 Specification

A specification π is a triple $(\mathcal{T}, \mathcal{D}, \mathcal{P})$ where $\mathcal{T}$ is the typing of base objects, $\mathcal{D}$ is the method definitions, and $\mathcal{P}$ is the parallel program calling the defined methods. Let Π denote the set of specifications. We will define each of the components in turn.

Typing. We define the set of object types, as follows. An object type is either a scalar or an array type. In an array type $st[\,]$, st is the scalar type of elements. A scalar type is either a basic, sequentially-consistent or linearizable type.

$ot \in OT$	::=	$st \mid st[\,]$	Object Type
$st \in ST$	::=	$bt \mid lt$	Scalar Type
$bt \in BT$	::=	$\{\textbf{BasicRegister}, \textbf{BasicSet}, \textbf{BasicMap}\}$	Basic Type
$ct \in SCT$	::=	$\{\textbf{CSRegister}\}$	Sequentially Consistent Type
$lt \in LT$	::=	$\{\textbf{AtomicRegister}, \textbf{AtomicCASRegister},$ $\textbf{Lock}, \textbf{TryLock}, \textbf{SCounter}, \textbf{SeqLock}\}$	Linearizable Type

Let Φ denote the set of base object names ϕ.

$$\phi \in \Phi \quad ::= \quad \{lock, reg, \ldots\} \qquad \text{Base Object Name}$$

The typing $\mathcal{T}$ is a mapping from base object names to object types. A comma-separated list of elements e is denoted as e^*.

$$\mathcal{T} \quad ::= \quad (\phi : ot)^* \qquad \text{Typing}$$

A thread-local object is an array that is indexed by the current thread identifier. A thread-local type is of the form **ThreadLocal** st and is a syntactic sugar for $st[\,]$.

Definitions and Program. We define the set of definitions $\mathcal{D}$ and programs $\mathcal{P}$. Let us define the set of values and variables first.

$$
\begin{array}{rcll}
i, v \in Val & ::= & \{1, 2, \ldots\} \cup \{true, false\} & \text{Value} \\
x \in ProgVar & ::= & \{i, r, \ldots\} & \text{Variable} \\
u \in U & ::= & x \mid v & \text{Variable or Value} \\
T \in Thread & ::= & \{1, 2, \ldots\} & \text{Thread Value} \\
t \in ThreadVar & ::= & \{t_1, t_2, \ldots\} & \text{Thread Variable} \\
\tau & ::= & t \mid T & \text{Thread Variable or Value} \\
n \in N & ::= & \{read, unlock, \ldots\} & \text{Method Name}
\end{array}
$$

The set of objects are defined as follows.

$$
\begin{array}{rcll}
o \in O & ::= & \phi \mid \phi[u] & \text{Shared Object} \\
\theta \in \Theta & ::= & o \mid \textbf{this} & \text{Object}
\end{array}
$$

$\phi[u]$ denotes uth element of array ϕ.

Let *Label* denote the set of labels c.

$$
\begin{array}{rcll}
c \in Label & ::= & \{doRead, doUnlock, \ldots\} & \text{Label}
\end{array}
$$

The set of definitions and programs are defined as follows:

$\mathcal{D} \in Defs$	::=	d^*	Definitions
$d \in Def$	::=	**def** $n_t(x^*)\ s, r$	Method Definition
$s \in Stmt$	::=	$s, s \mid$ **if** $b\ s$ **else** $s \mid q \mid x = u + u$	Statement
$b \in BCond$	::=	$u = u \mid u = u + u \mid u < u \mid \neg b \mid b \wedge b$	Condition
$q \in Call$	::=	$c \triangleright\ x = o.n_\tau(u^*) \mid c \triangleright$ **return** u	Call
$r \in Order$	::=	'{' $(c \rightarrow c)^*$ '}'	Order
$\mathcal{P} \in Prog$	::=	$p_0, (p_1 \| p_2 \| ... \| p_n)$	Parallel Program
$p \in TProg$	::=	$p; p \mid$ **if** $b\ p$ **else** $p \mid c \triangleright\ x = n_\tau(u^*)$	Sequential Program

The definition section $\mathcal{D}$ is a sequence of method definitions d. The method definition **def** $n_t(x^*)\ s, r$ defines a method named n with parameters t and x^* with the body s and the declared order r. The parameter t is the current thread identifier. It is written as a subscript as it is sometimes elided when it is not needed or is evident from the context. A statement s is either a sequence, a conditional, a base object method call, a return statement or a math operation. The statements s_1, s_2 and **if** $b\ s$ **else** s are sequencing and conditional statements. A condition b is a boolean expression on variables and values. In a method call $c \triangleright\ x = o.n_\tau(u^*)$, c is the label, n is the method name, o is the receiving object, τ the thread argument, u^* are the data arguments and x is the return variable. In a return statement $c \triangleright$ **return** u, c is the label and u is the returned value or variable. The semantics of the language will allow out-of-order execution of method calls. Any two labels that are left unordered by the specification may be reordered in the execution. Data and control dependencies in s impose execution order between statements. (Data and control dependencies are standard and defined more precisely in section 10.1.1.) The programmer can explicitly require additional orders for the body s of a defined method as the declared order r. The declared program order r is a binary relation on the set of labels of s. The orders imposed by locks can be declared in r.

The program section $\mathcal{P}$ is of the form $p_0, (p_1 \| p_2 \| ... \| p_n)$ where p_0 is the initialization program, and p_1, p_2, ... p_n are the parallel programs. A sequential program p is either a sequence, a conditional or a method call. In a method call $c \triangleright\ x = n_\tau(u^*)$, c is the label, n is the method name (that is defined in the method definitions), τ is the current thread argument, u^* are the data arguments and x is the return variable. The object **this** is the object of the current specification, is the default receiver object and so is elided.

Well-formedness. Consider a specification $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$ where $\mathcal{D} = d^*$ and $\mathcal{P} = p_0, (p_1 \| p_2 \| ... \| p_n)$. The specification satisfies the following well-formedness conditions that can be statically enforced. (0) The base name of every object o in $\mathcal{P}$ is typed in $\mathcal{T}$. (1) Every object is initialized in the execution of p_0. (2) Every branch of every method definition ends in a return statement. (3) The thread argument of each method call is the identifier of the thread in which it is called. (4) The array access index of every thread-local object is the current thread identifier. (5) Labels are unique. The names of the defined methods are unique. (6) Every variable is bound only once in the program. (7) For each method definition, the transitive closure of its data and control dependencies and the declared orders is acyclic.

Derived Specification. We define the function $basetype : OT \mapsto ST$ that maps a type to its base scalar type as follows: $basetype(st) = st$, and $basetype(st[]) = st$. We define the function base name $basename : \Theta \mapsto \Phi$ that maps an object name to its base name as follows: $basename(\mathbf{this}) = \mathbf{this}$, $basename(\phi) = \phi$, $basename(\phi[u]) = \phi$. Similarly, we define the function index $index : \Theta \mapsto U$ that maps an object name to its index as follows: $index(\mathbf{this}) = 0$, $index(\phi) = 0$, $index(\phi[u]) = u$.

Consider a specification $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$ where $\mathcal{D} = d^*$ and $\mathcal{P} = p_0, (p_1 \| p_2 \| ... \| p_n)$. We define the function $\mathcal{T}_{base}$ as follows: $\mathcal{T}_{base}(o) = basetype(\mathcal{T}(basename(o)))$. The names of methods defined in a program are unique. Thus, we define $par1_\pi : N \mapsto ProgVar$ that maps

method names to their first parameter. Similarly, $param2_\pi$ and $tpar_\pi$ are defined that map method names to their second parameter and current thread parameter. As the labels of a program are unique, we define the function $obj_\pi : Label \mapsto \Theta$ that maps the label of a method call to it receiver object. Similarly, the functions $index_\pi$, $name_\pi$, $thread_\pi$, $arg1_\pi$, $arg2_\pi$ and $retv_\pi$ map the label of a method call to the array index of the receiver object, the name of the method, the current thread argument, the first and second argument and the return variable of the method call. Let the *execution condition* of a statement be the conjunction of all of its enclosing if or else conditions. Let the function $cond_\pi : Label \mapsto BCond$ map the label of method calls to their execution condition. For conciseness, we treat return statements with the same notation as method calls. For a return statement, we let $name_\pi$ and $arg1_\pi$ map to $return$ and the argument of the return statement respectively. For example, consider the following method definition.

$$
\begin{array}{l}
\textbf{def } n_t(i, x) \\
\quad \textbf{if } (i > 1) \\
\quad\quad \textbf{if } (x < 2) \\
\quad\quad\quad c_1 \triangleright \ y = r[i].write_t(x), \\
\quad c_2 \triangleright \ \textbf{return } 1
\end{array}
$$

An if-then statement is a syntactic sugar for an if-then-else statement where the else branch is a call on a dummy object. For example above, we have $tpar_\pi(n) = t$, $par1_\pi(n) = i$, $param2_\pi(n) = x$, $obj_\pi(c_1) = r[i]$, $index_\pi(c_1) = i$, $name_\pi(c_1) = write$, $thread_\pi(c_1) = t$, $arg1_\pi(c_1) = x$, $retv_\pi(c_1) = y$, $cond_\pi(c_1) = (i > 1) \ \wedge \ (x < 2)$, $name_\pi(c_2) = return$ and $arg1_\pi(c_2) = 1$.

Let $Labels(s)$ denote the set of labels in s. Let $Labels_\pi(n)$ denote the set of labels in the body of n. Let $Returns_\pi(n)$ denote the set of labels of return statements in the body of n. Let $Labels(\mathcal{P})$ denote the set of labels in $\mathcal{P}$. Let $Labels(\pi)$ denote the set of labels in

π. Let $Calls_\pi(\phi, n)$ denote the set of labels of call statements where the method name n is called on the base object name ϕ. Let $PreReturns_\pi(c)$ denote the set of labels of the return statements before the statement labeled c in π. (The sets $Calls_\pi(\phi, n)$ and $PreReturns_\pi(c)$ are more precisely defined in section 10.1.1.)

Let $\rightarrow_n$ denote the irreflexive transitive closure of the data and control dependencies and the declared order of n. Let the program order $\rightarrow_\pi$ be the irreflexive partial order on $Labels(\pi)$ defined as the union of the following (1) the initialization order (that orders labels of p_0 before labels of parallel programs), (2) the sequential order of the sequential programs p_i (3) For each method definition n, the order $\rightarrow_n$.

2.2.2 TM Algorithm Specification

A transactional memory specification is $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$ where

$$\begin{aligned} \mathcal{D} &= \textbf{def } init_t()\ s_0, r_0, && (2.1)\\ &\quad\ \textbf{def } read_t(i)\ s_1, r_1,\\ &\quad\ \textbf{def } write_t(i, v)\ s_2, r_2,\\ &\quad\ \textbf{def } commit_t()\ s_3, r_3,\\ &\quad\ d^* \\ \mathcal{P} &= tran_0, (tran_1 \parallel tran_2 \parallel ... \parallel tran_n) && (2.2) \end{aligned}$$

Transactional memory is an object that encapsulates a set of locations. Each location i stores a value v. It has four methods $init_t()$, $read_t(i)$, $write_t(i, v)$ and $commit_t()$. The method call $init_t()$ initializes the transaction t. The method call $read_t(i)$ returns the value of location i or $\mathbb{A}$ (if the transaction is aborted). The method $write_t(i, v)$ writes v to location i and returns *ok* (if the operation is completed successfully) or returns $\mathbb{A}$ (if the transaction is aborted). The method $commit_t()$ tries to commit transaction t and returns $\mathbb{C}$ (if the transaction is successfully committed) or returns $\mathbb{A}$ (if it is aborted). The three

specific symbols $\mathbb{C}$, $\mathbb{A}$ and *ok* are returned in specification of transactional memory algorithms to denote commitment or abortion of the transaction and normal completion of a write operation respectively. These values are set aside to represent how the operation is completed and are not used as values of locations.

The initializing transaction $trans_0$ that initializes every location to zero is defined as follows:

$$\begin{aligned} trans_0 \quad := \quad & IL_0 \triangleright \ init_0(); \\ & c_{00} \triangleright \ write_0(0,0); \\ & c_{01} \triangleright \ write_0(1,0); \\ & ... \\ & c_{0m} \triangleright \ write_0(m,0); \\ & CL_0 \triangleright \ commit_0() \end{aligned} \tag{2.3}$$

Each transaction $trans_j$ $1 \leq j \leq n$ is defined as follows:

$$\begin{aligned} trans_j \quad := \quad & IL_j \triangleright \ init_j(); \\ & op_j \\ op_j \quad := \quad & c \triangleright \ x = read_j(v_1, v_2); \\ & \textbf{if } (\neg(x = \mathbb{A})) \\ & \quad op_j \\ | \quad & c \triangleright \ x = write_j(v); \\ & \textbf{if } (\neg(x = \mathbb{A})) \\ & \quad op_j \\ | \quad & CL_j \triangleright \ commit_j() \end{aligned} \tag{2.4}$$

Note that this dissertation does not consider non-transactional accesses and publication/privatization safety.

Well-formedness. The *init* method returns *ok*. The *read* method does not return *ok* or $\mathbb{C}$. The *write* method does not return $\mathbb{C}$. The *commit* method either returns $\mathbb{C}$ or $\mathbb{A}$.

$\forall c \in Returns_\pi(init)\colon arg1_\pi(c) = ok$

$\forall c \in Returns_\pi(read)\colon arg1_\pi(c) \neq ok \ \wedge \ arg1_\pi(c) \neq \mathbb{C}$

$\forall c \in Returns_\pi(write)\colon arg1_\pi(c) \neq \mathbb{C}$

$\forall c \in Returns_\pi(commit)\colon arg1_\pi(c) = \mathbb{C} \ \vee \ arg1_\pi(c) = \mathbb{A}$

In addition, it is assumed that in every execution of the transaction $trans_0$, all the *write* method calls return *ok*.

Let Π_{TM} denote the set of transactional memory specifications.

We define two functions *initOf* and *commitOf* that map a thread value to its initialization and commitment labels.

$$initOf(T) = IL_T \tag{2.5}$$

$$commitOf(T) = CL_T \tag{2.6}$$

2.2.3 Extended Syntax

The core syntax can be used to define expressive syntactic sugar.

As defined above, A thread-local type **ThreadLocal** *st* is a syntactic sugar for $st[\,]$.

The boolean expression $u \neq u'$ is a syntactic sugar for $\neg(u = u')$. The boolean expression b is a syntactic sugar for $b = true$. The statement **return** $(u = u')$ is a syntactic sugar for **if** $(u = u')$ **return** $true$ **else return** $false$.

A **return** statement without the return argument is used as a syntactic sugar for a return statement that returns a dummy value. An if-then statement is a syntactic sugar for an if-then-else statement where the else branch is a call on a dummy object.

As a convenience, the definition of methods may non-recursively call other defined methods. Note that these calls can always be inlined.

We now define **foreach** statement as a syntactic sugar. The **foreach** statement iterates over sets and maps.

Consider an object *set* of type set. The following **foreach** statement executes the statement s for each member i of *set*.

$$\begin{array}{l} c \triangleright \ \textbf{foreach} \ (i \in set) \\ \quad s \end{array} \tag{2.7}$$

Let b be a fresh variable name. We define $sIter(s, i)$, the ith iteration, as follows:

$$\begin{array}{rcl} sIter(s,i) & = & c_i \triangleright \ b_i = set.contains(i), \\ & & \textbf{if} \ (b_i) \\ & & \quad sIndexed(s,i) \end{array} \tag{2.8}$$

where $sIndexed(s, i)$ denotes a transformation of s where every label c is replaced by c_i and every variable x that is assigned in s is replaced by x_i. The **foreach** statement is a syntactic sugar for $sIters(s, 0)$ that is

$$\begin{array}{l} sIter(s,0), \\ sIter(s,1), \\ sIter(s,2), \\ ... \\ sIter(s,max) \end{array} \tag{2.9}$$

(where max is the maximum value) with the following declared order

$$\forall c \in Labels(sIter(s,i)), c' \in Labels(sIter(s,i+1)) \colon c \to c' \tag{2.10}$$

Similarly, consider an object map of type map. The following **foreach** statement executes the statement s for each mapping i to v in map.

$$\begin{array}{l} c \triangleright \ \textbf{foreach} \ ((i,v) \in map) \\ \qquad s \end{array} \tag{2.11}$$

We define $mIter(s,i)$, the $i$the iteration, as follows:

$$\begin{array}{rcl} mIter(s,i) & = & c_i \triangleright \ v_i = map.get(i), \\ & & \textbf{if} \ (v_i \neq \bot) \\ & & \qquad mIndexed(s,i) \end{array} \tag{2.12}$$

where $mIndexed(s,i)$ denotes a transformation of s where every label c is replaced by c_i, v is replaced with v_i, and every variable x that is assigned in s is replaced by x_i. We define $mIters(s,i)$, the sequence of iterations starting from iteration i, as follows:

$$\begin{array}{rcl} mIters(s,i) & ::= & mIter(s,i), \\ & & \qquad mIters(s,i+1) \end{array} \tag{2.13}$$

The **foreach** statement is a syntactic sugar for $mIters(0)$ that is

$$\begin{aligned} & mIter(s, 0), \\ & mIter(s, 1), \\ & mIter(s, 2), \\ & ... \\ & mIter(s, max) \end{aligned} \tag{2.14}$$

(where max is the maximum value) with the following declared order

$$\forall c \in Labels(mIter(s, i)), c' \in Labels(mIter(s, i+1)) \colon c \rightarrow c' \tag{2.15}$$

Now, we extend the syntax with records. A record type definition rt is defined as follows:

$$\begin{array}{rcll} rec \in Rec & ::= & \{Node, Locator, ...\} & \text{Record Type name} \\ rt \in RT & ::= & rec \text{ `\{'} (\phi_i : ot_i)^* \text{ `\}'} & \text{Record Type} \end{array}$$

The record type named rec is defined as a collection of fields ϕ_i of type ot_i. The set of statements is extended with the **new** and **clone** statements.

$$\begin{array}{rcll} s & ::= & ... & \\ & & c \triangleright\ x = \textbf{new}\ rec \text{ `('`)'} & \text{New Statement} \\ & & c \triangleright\ x = \textbf{clone}\ \text{ `('}x\text{`)'} & \text{Clone Statement} \end{array}$$

The **new** statement creates an instance of the record type and returns a reference to it. The **clone** statement creates a clone of the record referenced by the argument and returns

a reference to it. The set of objects is extended as follows

$$
\begin{array}{rcll}
o \in O & ::= & \phi \mid \phi[u] \mid & \text{Shared Object} \\
 & & x.\phi \mid x.\phi[u] &
\end{array}
$$

$x.\phi$ denotes the field named ϕ of the record that the variable x references.

2.2.4 Example Specifications

As example specifications, we present Dekker algorithm and TL2, a variant of TL2, visible and invisible reads versions of DSTM and McRT TM algorithms.

Dekker Algorithm Specification.

Dekker algorithm specified in Figure 2.1 provides mutual exclusion for two threads. It uses two atomic registers as flags. Using basic registers can lead to a race and violation of mutual exclusion. Each thread first sets its own flag and then reads the flag of the other thread. The order of writing the flag of the current thread and then reading the flag of the other thread is crucial to the correctness. Reordering these two accesses can violate mutual exclusion. A thread enters its critical region only if it finds the flag of the other thread unset. The type of the flags and the order of accesses to them are explicitly captured in the specification.

$\mathcal{T}$:		
f_1 : **AtomicRegister** f_2 : **AtomicRegister**		
$\mathcal{D}$:		
def $init()$	**def** $tryLock1()$	**def** $tryLock2()$
$W_{01} \triangleright \ f_1.write(0),$	$W_1 \triangleright \ f_1.write(1),$	$W_2 \triangleright \ f_2.write(1),$
$W_{02} \triangleright \ f_2.write(0),$	$R_2 \triangleright \ x_2 = f_2.read(),$	$R_1 \triangleright \ x_1 = f_1.read(),$
	if $(x_2 = 0),$	**if** $(x_1 = 0)$
	$C_{1t} \triangleright$ **return** $true$	$C_{2t} \triangleright$ **return** $true$
	else	**else**
	$C_{1f} \triangleright$ **return** $false,$	$C_{2f} \triangleright$ **return** $false,$
	$W_1 \rightarrow R_2,$	$W_2 \rightarrow R_1;$
$\mathcal{P} =$		
$L_0 \triangleright \ init(),$		
$L_1 \triangleright \ l_1 = tryLock1()$ $\quad \| \quad$ $L_2 \triangleright \ l_2 = tryLock2()$		

Figure 2.1: π_{Dekker} Dekker Algorithm Specification

TL2 Algorithm.

We specify TL2 algorithm [18] in Figure 2.2. Algorithm such as SwissTM [23] are optimizations of TL2.

Synchronization objects. TL2 algorithm uses the following synchronization objects: Value registers reg: an array of type basic register of size equal to the number of locations. Version registers ver: an array of type atomic register of size equal to the number of locations with the initial value 0. Locks $lock$: an array of type try-lock of size equal to the number of locations that are initially released. Global version clock $clock$: a strong counter with the initial value 0. Read version $rver$: a thread-local basic register. Read set $rset$: a thread-local basic set that is initially $\emptyset$. Write set $wset$: a thread-local basic map that is initially $\emptyset$. Lock set $lset$: a thread-local basic set that is initially $\emptyset$.

As observed by previous work, in the original paper, "the authors maintain the version number and the lock bit of every variable in the same memory word" [31], thus, the order of reading the lock and the version of read locations in the commit procedure is ambiguous. In our specification, we treat the lock and the version as separate registers and make the orders explicit.

Algorithm. TL2 is a deferred-update TM algorithm. A value that a transaction t writes to a location is buffered in the write set $wset[t]$ at $W01$ and is written back to register $reg[i]$ at $C16$ while t is committing. TL2 records a version in the register $ver[i]$ for the value stored in the register $reg[i]$. The version register $ver[i]$ is updated to ascending numbers at $C17$ after a new value is written back to $reg[i]$ at $C16$. The try-lock $lock[i]$ is used for exclusive access to location i. At commit, the lock $lock[i]$ of each location i in the write set $wset[t]$ is acquired at $C01$ to $C06$. (If a lock cannot be acquired, the previously acquired locks are released at $C05$ and the transaction is aborted.) Then, a new snapshot is read from $clock$ at $C07$. Then, for each location in the read set $rset[t]$, first $lock[i]$ and then $ver[i]$ are read at $C10$ and $C11$ and the read is validated. (If a read is not validated, the acquired locks are released at $C12$ and the transaction is aborted.) Finally, the values buffered in $wset[t]$

are written back at $C15$ to $C18$. For each pair in the write set $wset[t]$, the following three operations execute in order. First, the buffered value is written back to $reg[i]$, then $ver[i]$ is updated, and then $lock[i]$ is released. In the *init* method, each transaction t reads the current snapshot version from *clock* at $I01$ and writes it to the read version register $rver[t]$ at $I02$. The read version is read at $R07$ and $C08$ to validate the read values. To read a location i, a transaction reads $ver[i]$, $reg[i]$, $lock[i]$ and again $ver[i]$ in order at $R03 - R06$ and then validates the read. (If the validation fails, the transaction is aborted.) Finally, i is added to the read set $rset[t]$ and the read value is returned.

Note that although $reg[i]$ may be read and written by different transactions, it is declared as a basic register. The objects $lock[i]$ and $ver[i]$ rule out racy accesses to $reg[i]$. The lock $lock[i]$ prevents concurrent writes to $reg[i]$. If a read from $reg[i]$ executes concurrently with a write to it, reads from $ver[i]$ and the subsequent checks abort the read.

$\mathcal{T}$:

reg: **BasicRegister**[ ], $rver$: **ThreadLocal BasicRegister**,
ver: **AtomicRegister**[ ], $rset$: **ThreadLocal BasicSet**,
$lock$: **TryLock**[ ], $wset$: **ThreadLocal BasicMap**,
$clock$: **SCounter**, $lset$: **ThreadLocal BasicSet**

$\mathcal{D}$:

def $init_t()$
$I01 \triangleright$ $snap = clock.read()$,
$I02 \triangleright$ $rver[t].write(snap)$,
$I03 \triangleright$ **return** ok,

def $read_t(i)$
$R01 \triangleright$ $pv = wset[t].get(i)$,
if $(pv \neq \bot)$
$R02 \triangleright$ **return** pv,

$R03 \triangleright$ $s_1 = ver[i].read()$,
$R04 \triangleright$ $v = reg[i].read()$,
$R05 \triangleright$ $l = lock[i].read()$,
$R06 \triangleright$ $s_2 = ver[i].read()$,
$R07 \triangleright$ $sver = rver[t].read()$,
if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$
$R08 \triangleright$ **return** $\mathbb{A}$,

$R09 \triangleright$ $rver[t].add(i)$,
$R10 \triangleright$ **return** v,
$\{R03 \rightarrow R04, \ R04 \rightarrow R05, \ R05 \rightarrow R06\}$,

def $write_t(i, v)$
$W01 \triangleright$ $wset[t].put(i, v)$,
$W02 \triangleright$ **return** ok,

def $commit_t()$
$C01 \triangleright$ **foreach** $(i \in wset[t])$
$C02 \triangleright$ $locked = lock[i].trylock()$,
if $(\neg locked)$
$C03 \triangleright$ $lset.add(i)$
else
$C04 \triangleright$ **foreach** $(i \in lset)$
$C05 \triangleright$ $lock[i].unlock()$,
$C06 \triangleright$ **return** $\mathbb{A}$,

$C07 \triangleright$ $wver = clock.iaf()$,

$C08 \triangleright$ $sver = rver[t].read()$,
if $(wver \neq sver + 1)$
$C09 \triangleright$ **foreach** $(i \in rset[t])$
$C10 \triangleright$ $l = lock[i].read()$,
$C11 \triangleright$ $s = ver[i].read()$,
if $(\neg(\neg l \ \wedge \ s \leq sver))$
$C12 \triangleright$ **foreach** $(i \in lset)$
$C13 \triangleright$ $lock[i].unlock()$,
$C14 \triangleright$ **return** $\mathbb{A}$,

$C15 \triangleright$ **foreach** $((i, v) \in wset[t])$
$C16 \triangleright$ $reg[i].write(v)$,
$C17 \triangleright$ $ver[i].write(wver)$,
$C18 \triangleright$ $lock[i].unlock()$,

$C19 \triangleright$ **return** $\mathbb{C}$,
$\{C01 \rightarrow C07, \ C10 \rightarrow C11, \ C09 \rightarrow C15,$
$C16 \rightarrow C17, C17 \rightarrow C18\}$,

Figure 2.2: π_{TL2} TL2 Algorithm Specification

TL2 Variant Algorithm.

Based on the insight we obtained from studying TL2, we could come up with a variant of TL2 that by swapping two instructions in the commit procedure, removes reading a version and a conjunct from the validation check in the read method. The removal of some operations in our TL2 variant raises the question of whether the TL2 variant can outperform TL2 which can be a thread of future work.

TL2 variant algorithm is specified in Figure 2.3. Compared to the TL2 algorithm, the TL2 variant removes the first read of $ver[i]$, moves reading $lock[i]$ before reading $reg[i]$, removes the conjunct $s_1 = s_2$ from the validation in the read method and swaps the order of writes to $reg[i]$ and $ver[i]$ in the commit method.

$\mathcal{T}$:

reg: **BasicRegister**[ ],	$rver$: **ThreadLocal BasicRegister**,
ver: **AtomicRegister**[ ],	$rset$: **ThreadLocal BasicSet**,
$lock$: **TryLock**[ ],	$wset$: **ThreadLocal BasicMap**,
$clock$: **SCounter**,	$lset$: **ThreadLocal BasicSet**

$\mathcal{D}$:

def $init_t()$
$I01 \triangleright$ $snap = clock.read()$,
$I02 \triangleright$ $rver[t].write(snap)$,
$I03 \triangleright$ **return** ok,

def $read_t(i)$
$R01 \triangleright$ $pv = wset[t].get(i)$,
if $(pv \neq \bot)$
$R02 \triangleright$ **return** pv,

$R03 \triangleright$ $l = lock[i].read()$,
$R04 \triangleright$ $v = reg[i].read()$,
$R05 \triangleright$ $s = ver[i].read()$,
$R06 \triangleright$ $sver = rver[t].read()$,
if $(\neg(\neg l \ \wedge \ s \leq sver))$
$R07 \triangleright$ **return** $\mathbb{A}$,

$R08 \triangleright$ $rver[t].add(i)$,
$R09 \triangleright$ **return** v,
$\{R03 \rightarrow R04,\ R04 \rightarrow R05\}$,

def $write_t(i, v)$
$W01 \triangleright$ $wset[t].put(i, v)$,
$W02 \triangleright$ **return** ok,

def $commit_t()$
$C01 \triangleright$ **foreach** $(i \in wset[t])$
$C02 \triangleright$ $locked = lock[i].trylock()$,
if $(\neg locked)$
$C03 \triangleright$ $lset.add(i)$
else
$C04 \triangleright$ **foreach** $(i \in lset)$
$C05 \triangleright$ $lock[i].unlock()$,
$C06 \triangleright$ **return** $\mathbb{A}$,

$C07 \triangleright$ $wver = clock.iaf()$,

$C08 \triangleright$ $sver = rver[t].read()$,
if $(wver \neq sver + 1)$
$C09 \triangleright$ **foreach** $(i \in rset[t])$
$C10 \triangleright$ $l = lock[i].read()$,
$C11 \triangleright$ $s = ver[i].read()$,
if $(\neg(\neg l \ \wedge \ s \leq sver))$
$C12 \triangleright$ **foreach** $(i \in lset)$
$C13 \triangleright$ $lock[i].unlock()$,
$C14 \triangleright$ **return** $\mathbb{A}$,

$C15 \triangleright$ **foreach** $((i, v) \in wset[t])$
$C16 \triangleright$ $ver[i].write(wver)$,
$C17 \triangleright$ $reg[i].write(v)$,
$C18 \triangleright$ $lock[i].unlock()$,

$C19 \triangleright$ **return** $\mathbb{C}$,
$\{C01 \rightarrow C07,\ C10 \rightarrow C11,\ C09 \rightarrow C15,$
$C16 \rightarrow C17, C17 \rightarrow C18\}$,

Figure 2.3: $\pi_{TL2Variant}$ TL2 Variant Algorithm Specification

DSTM Algorithm.

Figure 2.4 shows the DSTM algorithm [39].

Synchronization objects. DSTM algorithm uses the following shared objects. *state* is an array of atomic cas registers of size equal to the number of threads. For each transaction t, $state[t]$ represents the state of t that is $\{\mathbb{R}, \mathbb{A}, \mathbb{C}\}$ (running, aborted or committed) with the default value $\mathbb{R}$. *start* is an array of atomic cas registers of size equal to the number of memory locations. For each memory location i, $start[i]$ stores a reference to an object of type *Loc* that represents the current state of location i. Let *Loc* be the class of objects with three fields: *writer*, *oldVal* and *newVal*. The field *writer* is a basic register that stores transaction identifiers. The two fields *oldVal* and *newVal* are basic registers that store values. DSTM allows only one writer to a location at a time. Therefore, two fields *oldVal* and *newVal* of *Loc* are sufficient to represent the values of a location. While the current writer transaction is writing to *newVal*, *oldVal* stores the stable value. The default value is a reference to a locator with *writer* set to T_0. The read set *rset* is a thread-local (transaction-local) basic set that stores pairs of index and value of read locations, and is $\emptyset$ initially.

Algorithm. DSTM is a deferred-update TM algorithm (The updates are delayed until the transaction commits [34]). Each location is represented as a reference to a record that stores the last and tentative states of the location. The current value of a location is decided according to the state of the last writer transaction to the location. Thus, a committing transaction updates the value of the locations that it has written to with a single cas on its own state.

Committing a transaction t takes effect by a cas on $state[t]$. During the commit, a transaction does not update the *oldVal* of the locations it has written to. Whether *oldVal* or *newVal* is the stable value is decided according to whether the state of the *writer* of the location is $\mathbb{C}$ or $\mathbb{A}$. To decide the stable value, if the state of the current *writer* is *Active*, it is *cas*ed to $\mathbb{A}$. Then, if the state of *writer* is $\mathbb{C}$, *newVal* is the stable value; if it is $\mathbb{A}$,

oldVal is the stable value. A new writer transaction of a location needs to set itself as the *writer* and also before overwriting *newVal*, if the state of the previous *writer* is $\mathbb{C}$, the new writer needs to copy *newVal* (that is the stable value) to *oldVal*. As *writer* and *oldVal* may be concurrently read by readers of the location, the new writer need to update them in isolation. Thus, the first write of a writer transaction instantiates a new *Loc* object with the current transaction as the *writer*, the stable value of the location as $OldVal$ and the value that it wants to write as *newVal*. Then, the new *Loc* object is installed in isolation by a *cas* on $start[i]$. On a global read, the pair of the read index and the read value is added to the read set $rset[t]$ that is validated before returning from a read or commit method call. Validation checks the equality of the logged value in $rset[t]$ to the last committed value of the location and that the transaction is not aborted by others.

DSTM is obstruction-free. A TM algorithm is obstruction-free if every transaction that runs without interleaving by other transactions makes progress. Obstruction-freedom precludes the use of locks. It is noticeable how DSTM avoids using locks in the following two parts of the algorithm. First, a committing transaction does not lock written locations. This is because a later read decides the stable value of a location according to the state of its last writer; therefore, the committing transaction does not need to lock the written locations to update their values but only updates its own state. Second, to write a new value to a location, the *writer* and *newVal* of the location are updated in isolation without acquiring locks. This is because DSTM employs a level of indirection. The two fields are updated atomically with a single cas on a reference.

$\mathcal{T}$:

Loc {
  writer: **BasicRegister**,
  oldVal: **BasicRegister**,
  newVal: **BasicRegister**
}

state: **AtomicCASRegister**[ ],
start: **AtomicCASRegister**[ ],
rset: **ThreadLocal** *BasicSet*

$\mathcal{D}$:

def $init_t()$
$I01 \triangleright$ $state[t].write(\mathbb{R})$,
$I02 \triangleright$ **return** *ok*,

def $read_t(i)$
$R01 \triangleright$ $s = state[t].read()$,
if $(s = \mathbb{A})$
$R02 \triangleright$ **return** $\mathbb{A}$,
$R03 \triangleright$ $st = start[i].read()$,
$R04 \triangleright$ $v = stableValue_t(st)$,
$R05 \triangleright$ $wr = st.writer.read()$,
if $(wr \neq t)$
$R06 \triangleright$ $rset[t].add(i, v)$,
$R07 \triangleright$ $valid = validate_t()$,
if $(\neg valid)$
$R08 \triangleright$ **return** $\mathbb{A}$,
$R09 \triangleright$ **return** v,
$\{R03 \rightarrow R07\}$

def $commit_t()$
$C01 \triangleright$ $valid = validate_t()$,
if $(\neg valid)$
$C02 \triangleright$ **return** $\mathbb{A}$,
$C03 \triangleright$ $b = state[t].cas(\mathbb{R}, \mathbb{C})$,
if (b)
$C04 \triangleright$ **return** $\mathbb{C}$
else
$C05 \triangleright$ **return** $\mathbb{A}$,
$\{C01 \rightarrow C03\}$

def $stableValue_t(st)$
$S01 \triangleright$ $t' = st.writer.read()$,
$S02 \triangleright$ $s' = state[t'].read()$,
if $(t' \neq t \ \wedge \ s' = \mathbb{R})$
$S03 \triangleright$ $state[t'].cas(\mathbb{R}, \mathbb{A})$,
$S04 \triangleright$ $s'' = state[t'].read()$,
if $(s'' = \mathbb{A})$
$S05 \triangleright$ $v = loc.oldVal.read()$
else
$S06 \triangleright$ $v = loc.newVal.read()$,
$S07 \triangleright$ **return** v,

def $write_t(i, v)$
$I01 \triangleright$ $s = state[t].read()$,
if $(s = \mathbb{A})$
$I02 \triangleright$ **return** $\mathbb{A}$,
$I03 \triangleright$ $st = start[i].read()$,
$I04 \triangleright$ $wr = st.writer.read()$,
if $(wr = t)$
$I05 \triangleright$ $st.newVal.write(v)$,
$I06 \triangleright$ **return** *ok*,
$I07 \triangleright$ $v' = stableValue_t(st)$,
$I08 \triangleright$ $st' = new\ Loc()$,
$I09 \triangleright$ $st'.writer.write(t)$,
$I10 \triangleright$ $st'.oldVal.write(v')$,
$I11 \triangleright$ $st'.newVal.write(v)$,
$I12 \triangleright$ $b = start[i].cas(st, st')$,
if (b)
$I13 \triangleright$ **return** *ok*,
else
$I14 \triangleright$ **return** $\mathbb{A}$,

def $validate_t()$
$V01 \triangleright$ **foreach** $((i, v) \in rset[t])$
$V02 \triangleright$ $st = start[i].read()$,
$V03 \triangleright$ $t' = st.writer.read()$,
$V04 \triangleright$ $s' = state[t'].read()$,
if $(s' = \mathbb{C})$
$V05 \triangleright$ $v' = loc.newVal.read()$
else
$V06 \triangleright$ $v' = loc.oldVal.read()$,
if $(v \neq v')$
$V07 \triangleright$ **return false**,
$V08 \triangleright$ $s = state[t].read()$,
$V09 \triangleright$ **return** $(s = \mathbb{R})$

Figure 2.4: π_{DSTM} DSTM Algorithm Specification

DSTM (visible reads) Algorithm.

Figure 2.5 presents DSTM (visible reads) algorithm [38].

Synchronization objects. DSTM (visible reads) algorithm uses the following shared objects. *state* is an array of atomic cas registers of size equal to the number of threads. For each transaction t, $state[t]$ represents the state of t that is $\{\mathbb{R}, \mathbb{A}, \mathbb{C}\}$ (running, aborted or committed) with the default value $\mathbb{R}$. *start* is an array of atomic cas registers of size equal to the number of memory locations. For each memory location i, $start[i]$ stores a reference to an object of type *Loc* that represents the current state of location i. Let *Loc* be the class of objects with four fields: *writer*, *rset*, *oldVal* and *newVal*. The field *writer* is a basic register that stores transaction identifiers. The field *rset* is a basic set that stores transaction identifiers. The two fields *oldVal* and *newVal* are basic registers that store values.

Algorithm. For each location, in addition to the two values $newVal$ and $oldVal$, the last *writer* transaction and the set *rset* of transactions that have read the location are recorded in the locator object. The read method reads the current locator, decides the stable value of the location according to the state of the latest *writer* of the location, creates a clone of the locator object, adds the current transaction to *rset* of the new locator and installs the new locator with a cas. The write method reads the current locator. If the current transaction is the current *writer* of the location, the new value is written to the $newVal$ field of the locator. Otherwise, a new locator object is created with the current value of the location as $oldVal$, the new value as $newVal$ and the current transaction as the *writer* and then the new locator is installed by a cas. Committing a transaction is done by as cas on its state.

The reader set of a location may be concurrently written by the readers of the location and read by the writers of the location. Therefore, readers and writers need to access the reader set in isolation. A reader-writer lock could be employed but would forfeit the obstruction-freedom property of the algorithm. The original algorithm, as specified here, uses the indirection mechanism. It stores the reader set in the locator object and achieves

isolation by a cas on the reference. RSTM [54], an optimization to DSTM, proposes a more efficient implementation for the reader set by a clever double-check in the reader transactions.

$\mathcal{T}$:

Loc {
  $writer$: **BasicRegister**,
  $rset$: **BasicSet**,
  $oldVal$: **BasicRegister**,
  $newVal$: **BasicRegister**,
}
$state$: **AtomicCASRegister**[ ],
$start$: **AtomicCASRegister**[ ]

$\mathcal{D}$:

def $init_t()$
$I01 \triangleright$ $state[t].write(\mathbb{R})$,
$I02 \triangleright$ **return** ok,

def $read_t(i)$
$R01 \triangleright$ $r = start[i].read()$,
$R02 \triangleright$ $v = currentValue_t(r)$,
$R03 \triangleright$ $r' = \textbf{clone}(r)$,
$R04 \triangleright$ $r'.rset.add(t)$,
$R05 \triangleright$ $b = start[i].cas(r, r')$,
$R06 \triangleright$ $s = state[t].read()$,
  if $(\neg b \ \vee \ (s = \mathbb{A}))$
$R07 \triangleright$ **return** $\mathbb{A}$
  else
$R08 \triangleright$ **return** v,
$\{R05 \rightarrow R06\}$

def $commit_t()$
$C01 \triangleright$ $b = state[t].cas(\mathbb{R}, \mathbb{C})$,
  if (b)
$C02 \triangleright$ **return** $\mathbb{C}$
  else
$C03 \triangleright$ **return** $\mathbb{A}$,

def $currentValue_t(r)$
$V01 \triangleright$ $t' = r.writer.read()$,
  if $(\neg(t' = t))$
$V02 \triangleright$ $state[t'].cas(\mathbb{R}, \mathbb{A})$,
$V03 \triangleright$ $s = state[t'].read()$,
  if $(s = \mathbb{A})$
$V04 \triangleright$ **return** $r.oldVal$
  else
$V05 \triangleright$ **return** $r.newVal$,

def $write_t(i, v)$
$W01 \triangleright$ $r = start[i].read()$,
$W02 \triangleright$ $w = r.writer.read()$,
  if $(w = t)$
$W03 \triangleright$ $r.newVal.write(v)$,
$W04 \triangleright$ **return** ok,

$W05 \triangleright$ $v' = currentValue_t(r)$,
$W06 \triangleright$ **foreach** $(t' \in r.rset)$
$W07 \triangleright$ $state[t'].cas(\mathbb{R}, \mathbb{A})$,

$W08 \triangleright$ $r' = new\ Loc()$,
$W09 \triangleright$ $r'.writer.write(t)$,
$W10 \triangleright$ $r'.oldVal.write(v')$,
$W11 \triangleright$ $r'.newVal.write(v)$,
$W12 \triangleright$ $b = start[i].cas(r, r')$,
  if (b)
$W13 \triangleright$ **return** ok,
  else
$W14 \triangleright$ **return** $\mathbb{A}$
$\{W06 \rightarrow W12\}$

Figure 2.5: $\pi_{DSTMVis}$ DSTM (visible reads) Algorithm Specification

McRT Algorithm.

Figure 2.6 shows the Core McRT algorithm.

Synchronization Objects. McRT uses the following synchronization objects. Value registers r: an array of basic registers of size equal to the number of memory locations. For each location i, $r[i]$ stores the value of location i. Version registers ver: an array of atomic registers of size equal to the number of memory locations. For each location i, $ver[i]$ stores the version for location i that is initially 0. Locks l: an array of try-locks of size equal to the number of memory locations. For each location i, $l[i]$ is initially released. The read set $rset$: a thread-local basic map from location indices to versions which is $\emptyset$ initially, and the undo set $uset$ a thread-local basic map from location indices to overwritten values which is $\emptyset$ initially.

In the original implementation, $ver[i]$ and $l[i]$ are stored in a single word. In our specification, we make the distinction explicit and specify the order of accesses to these registers. In addition, the original implementation overwrites the version bits with the transaction descriptor during the lock acquisition. Therefore, the versions had to be cached not only during the read method call but also during the write method call. Our specification stores only versions in the version registers and avoids caching of those registers during the write method call.

Algorithm. The first write to location i, tries to acquire $l[i]$ at $W02$ before writing to $r[i]$ at $W06$. McRT is a direct-update algorithm. It directly writes to $r[i]$ during the write method call before the commit method is invoked. Therefore, the old value of $r[i]$ is read and cached in the undo set $uset[t]$ at $W04-W05$ and restored to $r[i]$ while the transaction is aborting at $A02-A04$. A non-local read method call reads $ver[i]$ and then $l[i]$ at $R02-R03$. If $l[i]$ is locked, the transaction is aborted at $R04$. The first non-local read method call from a location caches the version in the read set $rset[t]$ at $R06$ which is used during the validation at $C01-C04$. For each read location i, the validation checks that the lock $l[i]$

is unlocked and the version $ver[i]$ is unchanged since it is read at $R02$. This ensures that $r[i]$ is unchanged since it is read at $R07$. For each written location, the version $ver[i]$ is incremented at $C08$ and the lock $l[i]$ is released at $C09$.

$\mathcal{T}$:

r: **BasicRegister**[ ],
ver: **AtomicRegister**[ ],
l: **TryLock**[ ],
$rset$: **ThreadLocal** $BasicMap$,
$uset$: **ThreadLocal** $BasicMap$

$\mathcal{D}$:

def $init_t()$
$I01 \triangleright$ **return** ok,

def $read_t(i)$
$R01 \triangleright$ $b = uset[t].contains(i)$,
if $(\neg b)$
$R02 \triangleright$ $rver = ver[i].read()$,
$R03 \triangleright$ $locked = l[i].read()$,
if $(locked)$
$R04 \triangleright$ **return** $abort_t()$,
$R05 \triangleright$ $pver = rset[t].get(i)$,
if $(pver = \perp)$
$R06 \triangleright$ $rset[t].put(i, rver)$,
$R07 \triangleright$ $v = r[i].read()$,
$R08 \triangleright$ **return** v
$\{R02 \rightarrow R03\}$

def $write_t(i, v)$
$W01 \triangleright$ $pval = uset[t].get(i)$,
if $(pval = \perp)$
$W02 \triangleright$ $locked = l[i].tryLock()$,
if $(\neg locked)$
$W03 \triangleright$ **return** $abort_t()$,
$W04 \triangleright$ $v' = r[i].read()$,
$W05 \triangleright$ $uset[t].put(i, v')$,
$W06 \triangleright$ $r[i].write(v)$,
$W07 \triangleright$ **return** ok,

def $commit_t()$
$C01 \triangleright$ **foreach** $((i, rver) \in rset[t])$
$C02 \triangleright$ $locked = l[i].read()$,
$C03 \triangleright$ $cver = ver[i].read()$,
if $(locked \vee rver \neq cver)$
$C04 \triangleright$ **return** $abort_t()$,

$C05 \triangleright$ **foreach** $(i \in uset[t]))$
$C06 \triangleright$ $cver = ver[i].read()$,
$C07 \triangleright$ $cver' = cver + 1$,
$C08 \triangleright$ $ver[i].write(cver')$,
$C09 \triangleright$ $l[i].unlock()$,
$C10 \triangleright$ **return** $\mathbb{C}$
$\{C02 \rightarrow C03, C08 \rightarrow C09\}$

def $abort_t()$
$A01 \triangleright$ **foreach** $((i, v) \in uset[t])$
$A02 \triangleright$ $r[i].write(v)$,
$A03 \triangleright$ $l[i].unlock()$,
$A04 \triangleright$ **return** $\mathbb{A}$
$\{A02 \rightarrow A03\}$

Figure 2.6: π_{McRT} McRT Algorithm Specification

NORec Algorithm.

$\mathcal{T}$:

$seqLock$: **SeqLock**,
reg: **BasicRegister**[ ]
$snap$: **ThreadLocal BasicRegister**,
$rset$: **ThreadLocal BasicMap**,
$wset$: **ThreadLocal BasicMap**,

$\mathcal{D}$:

def $init_t()$
do
$I01 \triangleright$ $(s, l) = seqLock.read()$
while (l),
$I02 \triangleright$ $snap[t] = s$,

def $read_t(i)$
$R01 \triangleright$ $pv = wset[t].get(i)$,
if $(pv \neq \perp)$
$R02 \triangleright$ **return** pv,
do
$R03 \triangleright$ $v = reg[i].read()$,
$R04 \triangleright$ $s1 = snap[t].read()$,
$R05 \triangleright$ $(s2, l2) = seqLock.read()$,
if $(s2 = s1 \ \wedge \ \neg l2)$
$R06 \triangleright$ **break**,
$R07 \triangleright$ $b = validate_t()$,
if $(\neg b)$
$R08 \triangleright$ **return** $\mathbb{A}$,
while $(true)$,
$R09 \triangleright$ $rset[t].put(i, v)$,
$R10 \triangleright$ **return** v,
$\{R03 \rightarrow R05\}$,

def $write_t(i, v)$
$W01 \triangleright$ $wset[t].put(i, v)$,
$W02 \triangleright$ **return** ok,

def $abort_t()$
$A01 \triangleright$ **return** $\mathbb{A}$

def $validate_t()$
$V01 \triangleright$ **while** $(true)$
do
$V02 \triangleright$ $(s1, l1) = seqLock.read()$,
while $(l1)$
foreach $((i, v) \in rset[t])$
$V03_i \triangleright$ $v' = reg[i].read()$,
if $(v \neq v')$,
$V04_i \triangleright$ **return** $false$,
$V05 \triangleright$ $(s2, l2) = seqLock.read()$,
if $(s2 = s1 \ \wedge \ \neg l2)$
$V06 \triangleright$ $snap[t].write(s1)$,
$V07 \triangleright$ **return** $true$,
$\{V02 \rightarrow V03_i, V03_i \rightarrow V05\}$,

def $commit_t()$
$C01 \triangleright$ $e = wset[t].isEmpty()$,
if (e)
$C02 \triangleright$ **return** $\mathbb{C}$,
do
$C03 \triangleright$ $s = snap[t].read()$,
$C04 \triangleright$ $d = seqLock.compareAndLock(s)$,
if (d)
$C05 \triangleright$ **break**,
$C06 \triangleright$ $b = validate_t()$,
if $(\neg b)$
return $\mathbb{A}$,
while $(true)$,
foreach $((i, v) \in wset[t])$
$C07_i \triangleright$ $reg[i].write(v)$,
$C08 \triangleright$ $seqLock.incAndUnlock()$,
$C09 \triangleright$ **return** $\mathbb{C}$
$\{C04 \rightarrow C07_i, C07_i \rightarrow C08\}$,

Figure 2.7: *NORec* NORec Algorithm Specification

2.3 Semantics

In this subsection, we define the semantics of specifications. We first define execution histories. Then, we define the semantics of base objects. Finally, we define the semantics of specifications as a set of execution histories.

2.3.1 Execution History

Now, we define execution histories and operations and relations on them.

Strings. If s_1 and s_2 are strings, we write $s_1 \Subset s_2$ iff s_1 is a subsequence of s_2. For example, $bd \Subset abcde$. Let s be an isogram (i.e. contains no repeating occurrence of the alphabet.) For any $s_1, s_2 \in s$, we write $s_1 \lhd_s s_2$ iff the last element of s_1 occurs before the first element of s_2 in s. For example $ab \lhd_{abcde} de$. We use $s(i)$ to denote the i^{th} element of s. We use $s \cdot s'$ to denote the concatenation of s and s'. For a set of strings $\{s_1 \dots s_n\}$, let $Interleave(s_1, \dots, s_n)$ denote the set of merges of s_1, ..., s_n.

Method calls and events. Let $LabelConst$ denote the set of labels l. The set of invocation events is $Inv = \{inv(l \triangleright\ o.n_T(u)) \mid l \in LabelConst, o \in O, n \in N, T \in Thread, u \in U\}$. The set of response events is $Res = \{ret(l \triangleright\ u) \mid l \in LabelConst, u \in U\}$. The set of events is $Ev = Inv \cup Res$. We will use the term completed method call to denote a sequence of an invocation event followed by the matching response event (with the same label). We use $l \triangleright\ v' = o.n_T(v)$ to denote the completed method call $inv(l \triangleright\ o.n_T(v)) \cdot ret(l \triangleright\ v')$.

Operations on event sequences. Let E and E' be event sequences. For a thread T, we use $E|T$ to denote the subsequence of all events of T in E. For an object o, we use $E|o$ to denote the subsequence of all events of o in E. $Sequential$ is the set of sequences of completed method calls.[1]

Execution history. An execution history X is a sequence of events where each invocation event has a unique label and every thread T is sequential (i.e. $X|T \in Sequential$).

[1]Note that we consider complete histories.

Let $History$ denote the set of execution histories. We say label l is in X and write $l \in X$ if there is an invocation event with label l in X. Let $Labels(X)$ denote the set of labels in X. Let $Threads(X)$ denote the set of threads in X. As the labels are unique in a history, the following functions on $Labels(X)$ are defined. The functions obj_X, $name_X$, $thread_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the thread identifier, the first and the second argument, and the return value associated with the labels. Similary, iEv and rEv functions on $Labels(X)$ map labels to the invocation and the response events associated with the labels.

A history X is equivalent to or indistinguishable from a history X', $X \equiv X'$, if one is a permutation of the other one that is only the events are reordered but the components of the events (including the argument and return values) are preserved.

Real-time relations. For an execution history X, we define the real-time relations $\prec_X$, $\preceq_X$, $\sim_X$, $\precsim_X$ on $Labels(X)$ as follows: First, $l_1 \prec_X l_2$ iff $rEv(l_1) \lhd_X iEv(l_2)$. $l_1 \preceq_X l_2$ iff $l_1 \prec_X l_2 \ \vee \ l_1 = l_2$. Second, $l_1 \sim_X l_2$ iff $l_1 \not\prec_X l_2 \ \wedge \ l_2 \not\prec_X l_1$. Third, $l_1 \precsim_X l_2$ iff $l_1 \prec_X l_2 \ \vee \ l_1 \sim_X l_2$.

From the definition of $Sequential$, we have that $X \in Sequential$ iff $\forall l, l' \in X : l \preceq_X l' \ \vee \ l' \prec_X l$. For an execution history X, we define the thread real-time relations $\lessdot_X$ and $\trianglelefteq_X$ as follows. First, $T \lessdot_X T'$ iff $X|T \lhd_X X|T'$. Second, $T \trianglelefteq_X T'$ iff $T \lessdot_X T' \ \vee \ T = T'$.

Now, we present a set of basic lemmas about execution orders. Please see the appendix Section 10.1.2.1 for proofs.

Lemma 1 (XASym)**.** *For every execution history X and method calls l and l', if $l \prec_X l'$ then $\neg(l' \prec_X l) \ \wedge \ \neg(l' \sim_X l) \ \wedge \ \neg(l' = l)$*

Lemma 2 (XTrans)**.** *For every execution history X and method calls l, $l.$ and l'', if $l \prec_X l'$ and $l' \prec l''$ then $l \prec_X l''$*

Lemma 3 (XXTrans)**.** *For every execution history X and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \precsim_X l_3$, and $l_3 \prec_X l_4$ then $l_1 \prec_X l_4$*

Lemma 4 (XTotal)**.** *For every execution history X and method calls l and l', if $l \in X$, and $l' \in X$, then $(l \prec_X l') \vee (l' \prec_X l) \vee (l \sim_X l') \vee (l = l')$*

Lemma 5 (X2X)**.** *For every execution history X and method calls l and l', if $l \prec_X l'$ then $l \in X$, and $l' \in X$.*

Lemma 6 (XI2X)**.** *For every execution history X and method calls l, l', and l'' if $l \prec_X l'$ and $inv(l') \lhd_X inv(l'')$ then $l \prec_X l''$.*

Lemma 7 (RX2X)**.** *For every execution history X and method calls l, l', and l'' if $ret(l) \lhd_X ret(l')$ and $l' \prec_X l''$ then $l \prec_X l''$.*

2.3.2 Synchronization Object Types

In this subsection, we first define the semantics of basic and linearizable objects. Then, we define the interface and the sequential specifications of the following abstract object types: register, lock, try-lock, counter, set and map. For each abstract object type, we define concrete synchronization object types. We define the following synchronization object types: basic register, atomic register, atomic cas register, lock, try-lock, strong counter, basic set and basic map. For each synchronization object type, we present lemmas that characterize the properties of its execution histories. Please see Section 10.1.2.2 for notes on the proof of the lemmas that we present in this subsection.[2]

Basic, Sequentially-consistent and Linearizable Object Types

The abstract type of each object o specifies the sequential specification of o, denoted by $SeqSpec(o)$, that is the prefix-closed set of correct sequential histories of o. In the following subsections, we will consider several synchronization object types and define their sequential specifications.

We consider three concurrent types: basic, sequentially-consistent and linearizable. Sequentially-consistent and linearizable objects comply with their sequential specification in every concurrent execution. Basic objects, on the other hand, comply with their sequential specification if they are accessed sequentially.

Definition 1 (Basic Object Semantics)**.** *Every sequential execution on a basic object is an execution in its sequential specification. The semantics of a basic object o,* $\mathbb{H}_B(o)$*, is a set of histories that is constrained as follows:*

$$\mathbb{H}_B(o) \cap Sequential \subseteq SeqSpec(o) \tag{2.16}$$

[2]In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l\colon p$ can be rewritten as $\bigwedge_{(l \in Labels(X))} p(X)$ and $\exists l\colon p$ can be rewritten as $\bigvee_{(l \in Labels(X))} p(X)$.

Definition 2 (Sequentially-consistent Object Semantics)**.** *An execution history X is sequentially-consistent for an object o iff there is an indistinguishable sequential history L that is in the sequential specification of o. L is a sequentialization and $\prec_L$ is a sequentialization order of X. The semantics of a sequentially-consistent object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and sequentialization pairs.*

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \ \wedge \ L \in SeqSpec(o) \ \wedge \ \forall T \in X : \ \prec_{X|T} \subseteq \prec_L\} \tag{2.17}$$

Definition 3 (Linearizable Object Semantics)**.** *An execution history X is linearizable for an object o iff there is an indistinguishable sequential history L that is in the sequential specification of o and is real-time-preserving. L is a linearization and $\prec_L$ is a linearization order of X. The semantics of a linearizable object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and linearization pairs.*

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \ \wedge \ L \in SeqSpec(o) \wedge \ \prec_X \subseteq \prec_L\} \tag{2.18}$$

Note that sequentially-consistent objects preserve execution order of method calls in the justifying sequential order only within threads while linearizable objects preserve it even across threads.

We now present lemmas for serialization and linearization orders.

Lemma 8 (X2L)**.** *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_X l'$ then $l \prec_L l'$.*

Lemma 9 (X2L')**.** *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_L l'$ then $l \precsim_X l'$.*

Lemma 10 (LASym)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $l \prec_L l'$ then $\neg(l' \prec_L l) \ \wedge \ \neg(l = l')$*

Lemma 11 (LTrans)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l, l', and l'', if $l \prec_L l'$ and $l' \prec_L l''$ then $l \prec_L l''$.*

Lemma 12 (LTotal)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $l \in X$ and $l' \in X$ then $(l \prec_L l') \vee (l' \prec_L l) \vee (l = l')$*

Lemma 13 (L2X)**.** *For every sequentialization or linearization L of an execution history X on object o and method calls l and l', if $(l \prec_L l')$ then $l \in X$, $l' \in X$, and l and l' are both on o.*

Lemma 14 (XLTrans)**.** *For every linearization L of an execution history X on object o and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \prec_L l_3$, $l_3 \prec_X l_4$, then $l_1 \prec_X l_4$*

See section 10.1.2.2 for proofs.

2.3.2.1 Register

Register. A register *reg* is an object that encapsulates a value and supports *read* and *write* methods. The method call $reg.read()$ returns the current encapsulated value of *reg*. The method call $reg.write(v)$ overwrites the encapsulated value of *reg* with v.

Definition 4. *The sequential specification of register reg is the set of sequential histories of read and write method calls on reg where every read returns the argument of the latest preceding write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) The sequential specification of*

a register r, $SeqSpec(r)$, is defined as follows:

$$isXRead_{X,r}(l_R) = l_R \in X \ \wedge \ obj_X(l_R) = r \ \wedge \ name_X(l_R) = read \quad (2.19)$$

$$isXWrite_{X,r}(l_W) = l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = write \quad (2.20)$$

$$NoWriteBetween_{X,r}(l_W, l_R) = \forall l'_W : isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_X l_W \ \vee \ l_R \prec_X l'_W) \quad (2.21)$$

$$isXWriter_{X,r}(l_W, l_R) = isXWrite_{X,r}(l_W) \ \wedge \quad (2.22)$$
$$l_W \prec_X l_R \ \wedge$$
$$NoWriteBetween_{X,r}(l_W, l_R)$$

$$Legal(r) = \{S \mid \forall l_R : isXRead_{S,r}(l_R) \Rightarrow \quad (2.23)$$
$$\exists l_W : isXWriter_{S,r}(l_W, l_R) \ \wedge$$
$$retv_S(l_R) = arg1_S(l_W)\}$$

$$SeqSpec(r) = \{S \mid S|r = S \ \wedge \ S \in Sequential \cap Legal(r)\} \quad (2.24)$$

Basic Register. A basic register is a basic instance of the register type.

Let $BasicRegister$ denote the type of basic registers.

Lemma 15. *In every sequential execution on a basic register, every read reads the value that the latest preceding write writes. Formally,*

$$\forall reg \in BasicRegister : \forall X \in \mathbb{H}_B(reg) : X \in Sequential \Rightarrow \quad (2.25)$$
$$\forall l_R : isXRead_{X,reg}(l_R) \Rightarrow$$
$$\exists l_W : isXWriter_{X,reg}(l_W, l_R) \ \wedge$$
$$retv_X(l_R) = arg1_X(l_W)$$

Two concurrent read method calls on a register do not conflict. Thus, basic registers can maintain consistency even when the execution involves concurrent read method calls. Let

us define

$$isXRaceFree_{X,r}(l) \;=\; \forall l_w \colon isXWrite_{X,r}(l_w) \Rightarrow l_w \preceq_X l \;\vee\; l \prec_X l_w \quad (2.26)$$

$$isXSequentiallyWritten_r(X) \;=\; \forall l \in X \colon isXWrite_{X,r}(l) \Rightarrow isRaceFree_{X,r}(l) \quad (2.27)$$

A method call is race-free if an only if there is no write method call that executes concurrent to it. An execution is sequentially-written if and only if every pair of write method calls on it are ordered in the execution order or in other words, every write method call on it is race-free.

Definition 5 (Basic Register Semantics)**.** *An execution history on a basic register is in the semantics of the basic register if and only if it is not sequentially-written or it is sequentially-written and every race-free read reads the value that the latest preceding write writes. The semantics of a basic register r, $\mathbb{H}_B(r)$, is defined as follows.*

$$\begin{aligned}\mathbb{H}_B(r) \;=\; \{X \mid X|o = X \;\wedge\; & \\ & isXSequentiallyWritten_r(X) \Rightarrow \\ & \forall l_r \colon isXRead_{X,r}(l_r) \;\wedge\; isXRaceFree_{X,r}(l_r) \Rightarrow \\ & \exists l_w \colon isXWriter_{X,r}(l_w, l_r) \;\wedge \\ & \quad retv_X(l_r) = arg1_X(l_w)\,\}\end{aligned} \quad (2.28)$$

Note that if an execution is not sequentially-written, reads may return arbitrary values. Similarly, racy reads may return arbitrary values.

Note that this definition satisfies the constraint of Definition 1.

Note that basic register models Lamport's notion of safe register [48].

Lemma 16 (BREG)**.** *In every sequentially-written execution on a basic register, every race-*

free read reads the value that the latest preceding write writes. Formally,

$$\begin{aligned}
&\forall reg \in BasicRegister\colon \forall X \in \mathbb{H}_B(reg)\colon isXSequentiallyWritten_r(X) \Rightarrow \quad (2.29)\\
&\forall l_R\colon isXRead_{X,reg}(l_R) \ \wedge \ isXRaceFree_{X,r}(l_R) \Rightarrow\\
&\exists l_W\colon isXWriter_{X,reg}(l_W, l_R) \ \wedge\\
&\qquad retv_X(l_R) = arg1_X(l_W)
\end{aligned}$$

Atomic Register. An atomic register is a linearizable instance of the register type.

Let *AtomicRegister* denote the type of atomic registers.

Let us define

$$\begin{aligned}
LNoWriteBetween_{X,L,r}(l_W, l_R) &= \forall l'_W\colon isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \ \vee \ l_R \prec_L l'_W) \quad (2.30)\\
isLWriter_{X,L,r}(l_W, l_R) &= isXWrite_{X,r}(l_W) \ \wedge \quad (2.31)\\
&\qquad l_W \prec_L l_R \ \wedge\\
&\qquad LNoWriteBetween_{X,L,r}(l_W, l_R)
\end{aligned}$$

Lemma 17 (AReg)**.** *In every execution on an atomic register, every read reads the value written by the last write linearized before it. Formally,*

$$\begin{aligned}
&\forall r \in AtomicRegister\colon \forall (X, L) \in \mathbb{H}_L(r)\colon \quad (2.32)\\
&\forall l_R\colon isXRead_{X,r}(l_R) \Rightarrow\\
&\exists l_W\colon isLWriter_{X,L,r}(l_W, l_R) \ \wedge\\
&\qquad retv_X(l_R) = arg1_X(l_W)
\end{aligned}$$

Sequentially-consistent Register. A sequentially-consistent register is a sequentially-consistent instance of the register type.

Let $CSRegister$ denote the type of atomic registers.

Consider the following four concurrent threads.

T_1	T_2	T_3	T_4
$L_{11} \triangleright \;\; r_1.write(1)$ $\parallel$	$L_{21} \triangleright \;\; r_2.write(1)$ $\parallel$	$L_{31} \triangleright \;\; x_1 = r_1.read()$ $\parallel$	$L_{41} \triangleright \;\; y_2 = r_2.read()$
		$L_{32} \triangleright \;\; y_1 = r_2.read()$	$L_{42} \triangleright \;\; x_2 = r_1.read()$
		$\{L_{31} \rightarrow L_{32}\}$	$\{L_{41} \rightarrow L_{42}\}$

If r_1 and r_2 are sequentially-consistent registers, there is an execution that results in the following values for the variables:

$x_1 = 1$, $y_1 = 0$, $y_2 = 1$ and $x_2 = 0$.

These values can be justified by the sequentialization order

(1) $L_{r_1} = L_{42} \triangleright \; x_2 = r_1.read() \;\cdot\; L_{11} \triangleright \; r_1.write(1) \;\cdot\; L_{31} \triangleright \; x_1 = r_1.read()$

for r_1 and the sequentialization order

(2) $L_{r_2} = L_{32} \triangleright \; y_1 = r_2.read() \;\cdot\; L_{21} \triangleright \; r_2.write(1) \;\cdot\; L_{41} \triangleright \; y_2 = r_2.read()$

for r_2.

If r_1 and r_2 are atomic registers, there is no execution that results in the values above for the variables. The real-time-preservation property precludes these executions. We assume that there is such an execution and show a contradiction. To have the above values for the variables, the linearization order of r_1 and r_2 should be as above in 1 and 2. By the program orders above, we have (3) $L_{31} \prec_X L_{32}$ (4) $L_{41} \prec_X L_{42}$. By X2L' on 2, we have (5) $L_{32} \precsim_X L_{41}$. By XXTrans on 3, 5 and 4, we have (6) $L_{31} \prec_X L_{42}$. By X2L on 6, we have $L_{31} \prec_{r_1} L_{42}$ that contradicts 1.

2.3.2.2 CAS (Compare-And-Swap) Register

A CAS register is an object that encapsulates a value and supports the *cas* method in addition to *read* and *write* methods. The method call $r.cas(v_1, v_2)$ updates the value of the register to v_2 and returns *true* if the current value of the register is v_1. It returns $false$

otherwise.

A *successful write* is either a *write* method call or a successful *cas* method call. The *written value* of a successful write is its first argument, if it is a *write* method call or is its second argument, if it is a *cas* method call.

Definition 6. *The sequential specification of cas register reg is the set of sequential histories of read, write and cas method calls on reg with the following two conditions. Every read returns the written value of the latest preceding successful write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) Every cas with the first argument v_1 returns true if the written value of the latest preceding successful write is v_1 and returns $false$ otherwise.*

Atomic CAS Register. An atomic CAS register is a linearizable instance of CAS register type.

Let *AtomicCASRegister* denote the type of Atomic CAS registers.

Let us define

$$isXCAS_{X,r}(l_W) = l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = cas \quad (2.33)$$

$$isXCWrite_{X,r}(l_W) = isXWrite(l_W) \ \vee \ (isXCAS(l_W) \ \wedge \ retv_X(l_W) = true) \quad (2.34)$$

$$writtenValue_X(l_W) = \begin{cases} arg1_X(l_W) & \text{if } name_X(l_W) = write \\ arg2_X(l_W) & \text{if } name_X(l_W) = cas \end{cases} \quad (2.35)$$

$$LNoWriteBetween_{X,L,r}(l_W, l_R) = \forall l'_W : isXCWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \ \vee \ l_R \prec_L l'_W) \quad (2.36)$$

$$isLCWriter_{X,L,r}(l_W, l_R) = isXCWrite_{X,r}(l_W) \ \wedge \quad (2.37)$$
$$l_W \prec_L l_R \ \wedge$$
$$LNoWriteBetween_{X,L,r}(l_W, l_R)$$

Lemma 18 (CASRegRead)**.** *In every execution on an atomic cas register, every read*

returns the value the last successful write linearized before it writes. Formally,

$$\begin{aligned}
&\forall r \in AtomicCASRegister \colon \forall (X, L) \in \mathbb{H}_L(r) \colon \\
&\forall l_R \colon isXRead_{X,r}(l_R) \Rightarrow \\
&\exists l_W \colon isLCWriter_{X,L,r}(l_W, l_R) \ \wedge \\
&\qquad retv_X(l_R) = arg1_X(l_W)
\end{aligned} \tag{2.38}$$

Lemma 19 (CASRegCAS)**.** *In every execution on an atomic cas register, every cas returns true if its first argument is equal to the argument of the last successful write linearized before it and returns $false$ otherwise. Formally,*

$$\begin{aligned}
&\forall reg \in AtomicCASRegister \colon \forall (X, Reg) \in \mathbb{H}_L(reg) \colon \\
&\forall l_C, l_W \colon \\
&\quad isXCAS_{X,reg}(l_C) \ \wedge \\
&\quad isLCWriter_{X,Reg,reg}(l_W, l_R) \\
&\Rightarrow \\
&\quad (writtenValue_X(l_W) = arg1_X(l_C) \Rightarrow retv_X(l_C) = true) \ \wedge \\
&\quad (\neg(writtenValue_X(l_W) = arg1_X(l_C)) \Rightarrow retv_X(l_C) = false)
\end{aligned} \tag{2.39}$$

2.3.2.3 Lock

Abstract lock. An abstract lock l is an object that encapsulates a state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: *lock*: The method call $l.lock()$ changes the state from $\mathbb{R}$ to $\mathbb{A}$. *unlock*: The method call $l.unlock()$ changes the state from $\mathbb{A}$ to $\mathbb{R}$. *read*: The method call $l.read()$ returns *true* if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method calls *lock* and *unlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 7. *The sequential specification of a lock l is the set of sequential histories L of lock, unlock, and read method calls on l where the sub-history of L for mutating methods is an alternating sequence of lock and unlock methods and every read method call in L returns true if the last mutating method call before it in L is a lock and returns false otherwise.*

Lock. A lock is a linearizable instance of the abstract lock type.

Let *Lock* denote the type of locks.

Now, we present some preliminary definitions and then lemmas about locks.

$$isXLock_{X,lo}(l) = \tag{2.40}$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = lock$$
$$isXUnlock_{X,lo}(l) = \tag{2.41}$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = unlock$$
$$isXRead_{X,lo}(l) = \tag{2.42}$$
$$l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = read$$

The common usage protocol for locks is that a thread unlocks a lock only if it has already acquired it. Many languages including Java enforce this property of programs by runtime checks. We capture this property as follows.

Definition 8. *A history is owner-respecting for a lock if every thread in the history releases*

the lock only after it has already acquired it.

$$
\begin{aligned}
&isXOwnerRespecting_{lo}(X) = \\
&\quad \forall l\colon isXUnlock_{X,lo}(l) \Rightarrow \\
&\quad \exists l'\colon isXLock_{X,lo}(l') \wedge \\
&\qquad thread_X(l') = thread_X(l) \wedge \\
&\qquad l' \prec_X l \wedge \\
&\qquad \forall l''\colon (isXUnLock_{X,lo}(l'') \ \wedge \ thread_X(l'') = thread_X(l)) \Rightarrow (l'' \prec_X l' \ \vee \ l \preceq_X l'')
\end{aligned}
\tag{2.43}
$$

Lemma 20. *If l is a lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of lock and unlock method calls by the same thread (possibly followed by a lock method call).*

Lemma 21 (Lock)**.** *In an owner-respecting execution for a lock l, if a lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1 is linearized before a lock method call by T_2. Formally,*

$$
\begin{aligned}
&\forall o \in Lock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u2}\colon \\
&\qquad (isXOwnerRespecting_o(X) \ \wedge \\
&\qquad isXLock_{X,o}(l_{l1}) \ \wedge \\
&\qquad isXUnlock_{X,o}(l_{u2}) \ \wedge \\
&\qquad l_{l1} \prec_L l_{u2}) \Rightarrow \\
&\quad \exists l_{u1}, l_{l2}\colon \\
&\qquad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\qquad isXLock_{X,o}(l_{l2}) \ \wedge \ thread_X(l_{l2}) = thread_X(l_{u2}) \ \wedge \\
&\qquad l_{u1} \prec_L l_{l2}
\end{aligned}
\tag{2.44}
$$

Lemma 22 (LockReadL)**.** *In an owner-respecting execution for a lock l, if a read method call that returns $false$ is linearized before an unlock method call by a thread T, then the read method call is linearized before a lock method call by T. Formally,*

$$\begin{aligned}
&\forall o \in Lock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{u1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\
&\quad l_{r2} \prec_L l_{u1}) \Rightarrow \\
&\exists l_{l1}\colon \\
&\quad isXLock_{X,o}(l_{l1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{r2} \prec_L l_{l1}
\end{aligned} \tag{2.45}$$

Lemma 23 (LockReadR)**.** *In an owner-respecting execution for a lock l, if a lock method call by a thread T is linearized before a read method call that returns $false$, then an unlock method call by T is linearized before the read method call. Formally,*

$$\begin{aligned}
&\forall o \in Lock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\
&\quad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}\colon \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{u1} \prec_L l_{r2}
\end{aligned} \tag{2.46}$$

Lemma 24 (LockReadM)**.** *In an owner-respecting execution for a lock l, every read method call that is linearized between a pair of matching lock and unlock method calls returns true. Formally,*

$$
\begin{aligned}
&\forall o \in Lock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X) \ \wedge \\
&\quad isXLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\
&\quad thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad \forall l'_{u1}\colon (isXUnlock_{X,o}(l'_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1} \ \vee \ l_{u1} \preceq_X l'_{u1}) \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \\
&\quad l_{l1} \prec_L l_{r2} \ \wedge \ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\quad retv_X(l_{r2}) = true
\end{aligned}
\tag{2.47}
$$

2.3.2.4 Try-lock

Abstract Try-lock. A try-lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and in addition to *lock*, *unlock* and *read* methods, it supports the *trylock* method. If the state of the *lock* is $\mathbb{R}$, $l.trylock()$ changes it to $\mathbb{A}$ and returns *true*. Otherwise, it returns $false$.

We call a *lock* method call or a successful *tryLock* method call, a *successful lock* method call. We call a *lock* method call, successful *tryLock* method call or *unlock* method call, a *mutating* method call.

Definition 9. *The sequential specification of a try-lock l is the set of sequential histories L of lock, unlock, read and tryLock method calls on l with the following conditions: The last mutating method call before a successful lock method call is an unlock method call. Similarly,*

the last mutating method call before an unlock method call is a successful lock method call. A tryLock method call returns true if the latest preceding mutating method call is an unlock and returns false otherwise. Similarly, A read method call returns true if the latest preceding mutating method call is a successful lock and returns false otherwise.

Try-Lock. A try-lock is a linearizable instance of the abstract try-lock type.

Let $TryLock$ denote the type of try-locks.

Similar to the *Lock* type, after some preliminary definitions, we define the owner-respecting histories and state the $TryLock$ type lemmas.

$$isXTryLock_{X,o}(l) = \tag{2.48}$$
$$l \in X \ \wedge \ obj_X(l) = o \ \wedge \ name_X(l) = tryLock$$
$$isXTLock_{X,o}(l) = \tag{2.49}$$
$$isXLock_{X,o}(l) \ \vee \ (isXTryLock_{X,o}(l) \ \wedge \ retv_X(l) = true)$$

The intuition for owner-respecting histories remains the same. A history is owner-respecting for a try-lock if every thread in the history releases the lock only after it has already acquired it. The minor difference from the prior definition for locks is that the acquisition of a try-lock is either by a *lock* method call or a successful *tryLock* method call.

$$isXTOwnerRespecting_o(X) = \tag{2.50}$$
$$\forall l\colon isXUnlock_{X,o}(l) \Rightarrow$$
$$\exists l'\colon isXTLock_{X,o}(l') \wedge$$
$$thread_X(l') = thread_X(l) \wedge$$
$$l' \prec_X l \wedge$$
$$\forall l''\colon (isXUnLock_{X,o}(l'') \ \wedge \ thread_X(l'') = thread_X(l)) \Rightarrow l'' \prec_X l' \ \vee \ l \preceq_X l''$$

Lemma 25. *If l is a try-lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of successful lock and unlock method calls by the same thread (possibly followed by a successful lock method call).*

Lemma 26 (TryLock)**.** *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1 is linearized before a successful lock method call by T_2. Formally,*

$$
\begin{aligned}
&\forall o \in TryLock \colon \forall (X, L) \in \mathbb{H}_L(o) \colon \forall l_{l1}, l_{u2} \colon \\
&\quad (isXTOwnerRespecting_o(X) \ \wedge \\
&\quad isXTLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXUnlock_{X,o}(l_{u2}) \ \wedge \\
&\quad l_{l1} \prec_L l_{u2}) \Rightarrow \\
&\exists l_{u1}, l_{l2} \colon \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad isXTLock_{X,o}(l_{l2}) \ \wedge \ thread_X(l_{l2}) = thread_X(l_{u2}) \ \wedge \\
&\quad l_{u1} \prec_L l_{l2}
\end{aligned}
\tag{2.51}
$$

Lemma 27 (TryLockReadL)**.** *In an owner-respecting execution for a try-lock l, a read method call that returns $false$ is linearized before if an unlock method call by a thread T*

then the read method call is linearized before a successful lock method call by T. Formally,

$$\begin{aligned}
&\forall o \in TryLock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{u1}, l_{r2}\colon \\
&\quad (isXTOwnerRespecting_o(X) \ \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \\
&\quad l_{r2} \prec_L l_{u1}) \Rightarrow \\
&\exists l_{l1}\colon \\
&\quad isXTLock_{X,o}(l_{l1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{r2} \prec_L l_{l1}
\end{aligned} \tag{2.52}$$

Lemma 28 (TryLockReadR)**.** *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T is linearized before a read method call that returns $false$, then an unlock method call by T is linearized before the read method call. Formally,*

$$\begin{aligned}
&\forall o \in TryLock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{r2}\colon \\
&\quad (isXTOwnerRespecting_o(X) \ \wedge \\
&\quad isXTLock_{X,o}(l_{l1}) \ \wedge \\
&\quad isXRead_{X,o}(l_{r2}) \ \wedge \ retv_X(l_{r2}) = false \\
&\quad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}\colon \\
&\quad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\
&\quad l_{u1} \prec_L l_{r2}
\end{aligned} \tag{2.53}$$

Lemma 29 (TryLockReadM)**.** *In an owner-respecting execution for a try-lock l, every read method call that is linearized between a pair of matching successful and unlock method*

calls returns true. Formally,

$$\begin{aligned}
&\forall o \in TryLock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\quad (isXOwnerRespecting_o(X)\ \wedge \\
&\quad isXTLock_{X,o}(l_{l1})\ \wedge \\
&\quad isXUnlock_{X,o}(l_{u1})\ \wedge \\
&\quad thread_X(l_{l1}) = thread_X(l_{u1})\ \wedge \\
&\quad \forall l'_{u1}\colon (isXUnlock_{X,o}(l'_{u1})\ \wedge\ thread_X(l_{l1}) = thread_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1}\ \vee\ l_{u1} \preceq_X l'_{u1}) \\
&\quad isXRead_{X,o}(l_{r2})\ \wedge \\
&\quad l_{l1} \prec_L l_{r2}\ \wedge\ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\quad retv_X(l_{r2}) = true
\end{aligned} \tag{2.54}$$

2.3.2.5 Sequence-lock

Abstract seq-lock. A seq-lock l is an object that encapsulates a number and an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$. It supports the *read*, *compareAndLock* and *incAndUnlock* methods. The method call $l.read()$ returns the pair of the encapsulated number and *true* if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method call $l.compareAndLock(n)$ compares the the encapsulated number with n and if they are equal, changes the state from $\mathbb{R}$ to $\mathbb{A}$ and returns *true*. Otherwise, it does not change the state of the seq-lock and returns $false$. The method call $l.incAndUnlock()$ increments the encapsulated number and changes the state from $\mathbb{A}$ to $\mathbb{R}$.

A successful *compareAndLock* and *incAndUnlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 10. *The sequential specification of a seq-lock l is the set of sequential histories L of read,*

compareAndLock, and incAndUnlock method calls on l with the following conditions:

Every read method call returns the pair of the number of incAndUnlock method calls before it and true if the last mutating method call before it is a successful compareAndLock and false otherwise.

A compareAndLock method call returns true if the last mutating method call before it is an incAndUnlock method call and the number of incAndUnlock method calls before it is equal to its argument. It returns false otherwise.

The last mutating method call before an incAndUnlock method call is a successful compareAndLock method call.

Seq-Lock. A seq-lock is a linearizable instance of the abstract seq-lock type.

Let *SeqLock* denote the type of seq-locks.

2.3.2.6 Counter

Abstract Counter: A counter c is an object that encapsulates a number and supports the following two methods: The method call $c.read()$ returns the current value of c. The method call $c.iaf()$ increments the value of c and returns the incremented value.

Definition 11. *The sequential specification of a counter c is the set of sequential histories of read and iaf method calls on c where every method call returns the number of iaf method calls before it (including the method call itself). Note that it is assumed that the initial value of the counter is zero.*

Strong Counter. A strong counter is a linearizable instance of abstract counter type.

Let *SCounter* denote the type of strong counters.

Lemma 30 (SCounter)**.** *The return value of every method call that is linearized before an*

iaf method call is smaller than the return value of the iaf method call. Formally,

$$\begin{aligned}
&\forall c \in SCounter\colon \forall (X, C) \in \mathbb{H}_L(c)\colon \forall l, l'\colon \\
&\quad l \in X \ \wedge\ l' \in X \ \wedge\ name_X(l') = iaf \ \wedge\ l \prec_C l' \\
&\Rightarrow \\
&\quad retv_X(l) < retv_X(l')
\end{aligned} \tag{2.55}$$

2.3.2.7 Set

A set s is an object that represents a set of values and supports the following methods: *add*: The method call $s.add(v)$ adds value v to set s. *contains*: The method call $s.containts(v)$ returns *true* if v is a member of s and $false$ otherwise.

Definition 12. *The sequential specification of a set s is the set of sequential histories of add and contains method calls on s where every contains method call returns true if there is a preceding add method call with the same argument, and returns $false$ otherwise. Note that it is assumed that the set is initially empty.*

Basic Set. A basic set is a basic instance of set type.

Let $BasicSet$ denote the type of basic sets.

Let us define

$$\begin{aligned}
&isXContains_{X,s}(l) = \\
&\quad l \in X \ \wedge\ obj_X(l) = s \ \wedge\ name_X(l) = contains
\end{aligned} \tag{2.56}$$

$$\begin{aligned}
&isXAdd_{X,s}(l) = \\
&\quad l \in X \ \wedge\ obj_X(l) = s \ \wedge\ name_X(l) = add
\end{aligned} \tag{2.57}$$

Lemma 31 (BasicSetContains)**.** *In every sequential execution on a basic set, for every*

contains method call that returns true, there is a preceding add method call with the same argument. Formally,

$$\begin{aligned}
&\forall s \in BasicSet\colon \forall X \in \mathbb{H}_B(s)\colon X \in Sequential \Rightarrow \\
&\forall l_c\colon isXContains_{X,s}(l_c) \ \wedge \ retv_X(l_c) = true \Rightarrow \\
&\exists l_a\colon isXAdd_{X,s}(l_a) \ \wedge \\
&\qquad arg1(l_a) = arg1(l_c) \ \wedge \ l_a \prec_X l_c
\end{aligned} \tag{2.58}$$

Lemma 32 (BasicSetAdd)**.** *In every sequential execution on a basic set, every contains method call that succeeds an add method call with the same argument returns true. Formally,*

$$\begin{aligned}
&\forall s \in BasicSet\colon \forall X \in \mathbb{H}_B(s)\colon X \in Sequential \Rightarrow \\
&\forall l_c, l_a\colon \\
&\qquad isXContains_{X,s}(l_c) \ \wedge \\
&\qquad isXAdd_{X,s}(l_a) \ \wedge \\
&\qquad arg1(l_a) = arg1(l_c) \ \wedge \ l_a \prec_X l_c \\
&\Rightarrow \\
&\qquad retv_X(l_c) = true
\end{aligned} \tag{2.59}$$

2.3.2.8 Map

A map m is an object that represents a mapping from a set of keys to a set of values and supports the following methods: *put*: The method call $m.put(k, v)$ adds or updates the mapping of the key k to the value v ($v \neq \perp$) in the map m. *get*: The method call $m.get(k)$ returns the value that the map m associates with the key k. It returns $\perp$ if m does not map k.

Definition 13. *The sequential specification of a map m is the set of sequential histories of*

put and get method calls on m where every get method call returns $\perp$ if there is no preceding put method call with the same key argument; otherwise it returns the second argument of the latest preceding put method call with the same key argument. Note that it is assumed that the map is initially empty.

Basic Map. A basic set is a basic instance of map type.

Let $BasicMap$ denote the type of basic maps.

Let us define

$$isXGet_{X,m}(l) = \tag{2.60}$$
$$l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = get$$
$$isXPut_{X,m}(l) = \tag{2.61}$$
$$l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = put$$
$$isXPutter_{X,m}(l_p, l_g) \Leftrightarrow \tag{2.62}$$
$$isXPut_{X,m}(l_p) \ \wedge \ arg1_X(l_p) = arg1_X(l_g) \ \wedge \ l_p \prec_X l_g \wedge \tag{2.63}$$
$$\forall l'_p \colon isXPut_{X,m}(l'_p) \ \wedge \ arg1_X(l'_p) = arg1_X(l_g) \Rightarrow (l'_p \preceq_X l_p \ \vee \ l_g \prec_X l'_p) \tag{2.64}$$

Lemma 33 (BasicMapGet)**.** *In every sequential execution on a basic map, the return value of every get method call that does not return $\perp$ is equal to the value argument of the latest preceding put method call with the same key argument. Formally,*

$$\forall m \in BasicMap \colon \forall X \in \mathbb{H}_B(m) \colon X \in Sequential \Rightarrow \tag{2.65}$$
$$\forall l_g \colon isXGet_{X,m}(l_g) \ \wedge \ \neg(retv_X(l_g) = \perp) \Rightarrow$$
$$\exists l_p \colon isPutter_{X,m}(l_p, l_g) \ \wedge$$
$$arg2_X(l_p) = retv_X(l_g)$$

Lemma 34 (BasicMapPut)**.** *In every sequential execution on a basic map, for every get*

method call g, if p is the latest preceding put method call with the same key argument then the return value of g is equal to the value argument of p. Formally,

$$\begin{aligned}
&\forall m \in BasicMap\colon \forall X \in \mathbb{H}_B(m)\colon X \in Sequential \Rightarrow \\
&\forall l_g, l_p\colon \\
&\quad isXGet_{X,m}(l_g) \ \wedge \\
&\quad isPutter_{X,m}(l_p, l_g) \ \wedge \\
&\Rightarrow \\
&\quad retv_X(l_g) = arg2_X(l_p)
\end{aligned} \tag{2.66}$$

2.3.3 History Semantics

In this subsection, we define the history semantics of specifications. It is a denotational semantics that defines the set of histories of a specification with a set of constraints enforcing the structure of the program and the guarantees of the base objects. Based on this denotational semantics, later chapters present a technique that can find bugs by constraint solving. The history semantics has the following three properties

- The semantics is *compositional* for base objects. It is defined abstractly from specific base object types. The semantics can be modularly augmented with new object types. The semantics of basic and linearizable objects are separately defined.
- The semantics models *true concurrency*. In contrast to interleaving semantics where method calls are the elements of histories, true concurrency considers a pair of invocation and response events for each method call. Therefore, concurrent execution of method calls is modeled.
- The semantics models *relaxed execution*. Methods that are not required to be ordered by the specification are allowed to execute out of order.

First, we present some preliminary definitions. The prefixing operator ' prefixes a program label before another program label or a program variable. The identity element for the prefixing operator is ϵ, thus $c = \epsilon\text{'}c$; $x = \epsilon\text{'}x$; $t = \epsilon\text{'}t$. We define the set of labels as follows:

$$\begin{array}{rcll} \varsigma & := & c \mid \epsilon & \text{Prelabel} \\ l \in \mathit{LabelConst} & ::= & \varsigma\text{'}c & \text{Constant Label} \end{array}$$

Each label l is of the form ς'c. ς is called the pre-label and c is called the leading label of l. A label c_1'c_2 is a call string that denotes a method call labeled c_2 in a *this* method called from a call site labeled c_1. On the other hand, the label c_1 denotes a *this* method call labeled c_1 in the concurrent program part. We extend the grammar of variables with labeled variables

Let $Labels_\pi(n) = \{\overline{c_i}\}, Returns_\pi(n) = \{\overline{c_r}\}$.

$$
\begin{aligned}
&[\![c \triangleright\ x' = n_\tau(u)]\!] = \{(X,\sigma) \mid \\
&\quad X = inv(c \triangleright\ n_\tau(u)) \cdot X' \cdot ret(c \triangleright\ x') \wedge \\
&\quad X' \in Sequential \wedge \\
&\quad \sigma(c\text{'}tpar_\pi(n)) = \sigma(\tau)\ \wedge\ \sigma(c\text{'}par1_\pi(n)) = \sigma(u) \wedge \\
&\quad Labels(X') \subseteq \{\overline{c\text{'}c_i}\} \wedge \\
&\quad \forall c\text{'}c_i \in X' : \\
&\qquad obj_{X'}(c\text{'}c_i) = c\text{'}obj_\pi(c_i)\ \wedge\ thread_{X'}(c\text{'}c_i) = c\text{'}thread_\pi(c_i) \wedge \\
&\qquad name_{X'}(c\text{'}c_i) = name_\pi(c_i)\ \wedge\ arg1_{X'}(c\text{'}c_i) = c\text{'}arg1_\pi(c_i) \wedge \\
&\qquad retv_{X'}(c\text{'}c_i) = c\text{'}retv_\pi(c_i) \wedge \\
&\quad \forall c_i \in \{\overline{c_i}\} : c\text{'}c_i \in X' \Leftrightarrow \\
&\qquad\qquad (\sigma(c\text{'}cond_\pi(c_i))\ \wedge\ \forall c_j \in PreReturns_\pi(c_i) \Rightarrow \neg c\text{'}c_j \in X')\ \wedge \\
&\quad \forall c_i, c_j \in \{\overline{c_i}\} : ((c_i \rightarrow_n c_j)\ \wedge\ c\text{'}c_i \in X'\ \wedge\ c\text{'}c_j \in X') \Rightarrow c\text{'}c_i \preceq_{X'} c\text{'}c_j \wedge \\
&\quad \forall c_r \in \{\overline{c_r}\} : c\text{'}c_r \in X' \Rightarrow \sigma(x') = \sigma(c\text{'}arg1_\pi(c_r))\}
\end{aligned}
\tag{2.67}
$$

$$
[\![p_1;\ p_2]\!] = \{(X,\sigma) \mid \exists X_1, X_2 : (X_1,\sigma) \in [\![p_1]\!]\ \wedge\ (X_2,\sigma) \in [\![p_2]\!]\ \wedge\ X = X_1 \cdot X_2\}
\tag{2.68}
$$

$$
[\![\textbf{if}\ b\ p_1\ \textbf{else}\ p_2]\!] = \{(X,\sigma) \mid ((X,\sigma) \in [\![p_1]\!]\ \wedge\ \sigma(b))\ \vee\ ((X,\sigma) \in [\![p_2]\!]\ \wedge\ \neg\sigma(b))\}
\tag{2.69}
$$

Let $\mathcal{P} = p_0, (p_1 \| p_2 \| ... \| p_n)$.

$$
\begin{aligned}
&[\![\pi]\!] = \{(X,\sigma,\mathcal{L}) \mid \\
&\quad \forall i \in \{0..n\} : \exists X_i : (X_i,\sigma) \in [\![p_i]\!]\ \wedge\ X' \in Interleave(X_1,...,X_n)\ \wedge\ X = X_0 \cdot X' \wedge \\
&\quad X'' = \sigma(X) \wedge \\
&\quad \forall o : \mathcal{T}_{base}(o) \in LT \cup SCT \Rightarrow (X''|o, \mathcal{L}(o)) \in \mathbb{H}_L(o) \wedge \\
&\quad \forall o : \mathcal{T}_{base}(o) \in BT \Rightarrow X''|o \in \mathbb{H}_B(o)\}
\end{aligned}
\tag{2.70}
$$

$$
\mathbb{H}(\pi) = \{X' \mid \exists (X,\sigma,\mathcal{L}) \in [\![\pi]\!]\ \wedge\ X' = \sigma(X)\}
\tag{2.71}
$$

Figure 2.8: History Semantics $\mathbb{H}(\pi)$ of a specification $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

as follows.

$$\begin{array}{rcll} x \in ProgVar & ::= & \{i, r, \ldots\} \mid \varsigma\text{'}x & \text{Variable} \\ t \in ThreadVar & ::= & \{t_1, t_2, \ldots\} \mid \varsigma\text{'}t & \text{Thread Variable} \end{array}$$

Labeled variables are used to denote local variables of *this* method calls. For example, c'x denotes the local variable x insider a *this* method call labeled c. Similarly, the prefixing operator is lifted to expressions such that every label and variable in the expression is prefixed. For example $c'(x_1 > x_2) = (c'x_1 > c'x_2)$.

Let σ denote a mapping from variables to concrete values. The history $\sigma(X)$ is the history that is obtained from the history X by substituting every variable x and t in X with $\sigma(x)$ and $\sigma(t)$ respectively. Similarly, the condition $\sigma(b)$ is the condition that is obtained by substituting every variable x and t of b with $\sigma(x)$ and $\sigma(t)$ respectively. Similar is the definition of $\sigma(o)$, $\sigma(u)$ and $\sigma(\tau)$.

Consider a specification $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$. The history semantics of π, $\mathbb{H}(\pi)$, is defined in Figure 2.8. We illustrate the definitions in the following paragraphs.

The semantics of a *this* method call is defined in Equation 2.67. Consider a method call

$$c \triangleright\ x' = n_\tau(u)$$

Every execution history X of a *this* method call starts with the invocation of n and finishes with a response from n. The enclosed history X' is an execution history of the body of n.

$$X = inv(c \triangleright\ n_\tau(u)) \cdot X' \cdot ret(c \triangleright\ x')$$

Each threads is sequential. Thus, the execution history X' of the body of n is a sequential execution history.

$$X' \in Sequential$$

A method can be called several time in the program. To have unique variable names, every variable (including the parameters) of the method is prefixed by the caller label. For example, c'x represents the variable x inside the method call labeled c. The function σ represents the mapping from variables to values at the end of the execution.

As defined before, $tpar_\pi(n)$ and $par1_\pi(n)$ are the thread parameter and the first parameter of the method n respectively. In the method call c defined above, τ and u are the thread argument and the first argument respectively. In every method call, the parameters are equal to the arguments.

$$\sigma(c\text{'}tpar_\pi(n)) = \sigma(\tau) \ \wedge \ \sigma(c\text{'}par1_\pi(n)) = \sigma(u)$$

Let the set of labels of method n, $Labels_\pi(n)$, be $\{\overline{c_i}\}$. Obviously, an execution of the body involves only the labels that are in the body itself. Note that to have unique labels, the labels c_i are prefixed by the caller label c.

$$Labels(X') \subseteq \{\overline{c\text{'}c_i}\}$$

If a method call labeled c_i is executed inside a *this* call labeled c, every variable in it is prefixed by c in the execution history. For example, a method call

$$c_i \triangleright \ y = \phi[i].n_t(x)$$

executed inside a *this* method call labeled c appears as follows in the execution history.

$$inv(c\text{'}c_i \triangleright \ \phi[c\text{'}i].n_{c\text{'}t}(c\text{'}x)) \cdot ret(c\text{'}c_i \triangleright \ c\text{'}y)$$

Thus, the components of every executed label $c\text{'}c_i$ are the components of c_i in the program prefixed with c.

$$\begin{aligned}
&\forall c\text{'}c_i \in X' : \\
&\quad obj_{X'}(c\text{'}c_i) = c\text{'}obj_\pi(c_i) \ \wedge \ thread_{X'}(c\text{'}c_i) = c\text{'}thread_\pi(c_i) \wedge \\
&\quad name_{X'}(c\text{'}c_i) = name_\pi(c_i) \ \wedge \ arg1_{X'}(c\text{'}c_i) = c\text{'}arg1_\pi(c_i) \wedge \\
&\quad retv_{X'}(c\text{'}c_i) = c\text{'}retv_\pi(c_i) \wedge
\end{aligned}$$

A labeled statement c_i is executed if and only if its condition $cond_\pi(c_i)$ is satisfied and no return statement before it is already executed.

$$\begin{aligned}
&\forall c_i \in \{\overline{c_i}\} : c\text{'}c_i \in X' \Leftrightarrow \\
&\qquad (\sigma(c\text{'}cond_\pi(c_i)) \ \wedge \ \forall c_j \in PreReturns_\pi(c_i) \Rightarrow \neg c\text{'}c_j \in X')
\end{aligned}$$

The execution order preserves the program order. If two labels are required to be ordered by the specification, they are ordered in the execution.

$$\forall c_i, c_j \in \{\overline{c_i}\} : \ ((c_i \rightarrow_n c_j) \ \wedge \ c\text{'}c_i \in X' \ \wedge \ c\text{'}c_j \in X') \Rightarrow c\text{'}c_i \preceq_{X'} c\text{'}c_j$$

Every execution of the body executes a return statement and the argument of the return statement is equal to the return value of the *this* method call. Let the set of return statements of the method n, $Returns_\pi(n)$, be $\{\overline{c_r}\}$.

$$\exists c_r \in \{\overline{c_r}\} : c\text{'}c_r \in X' \ \wedge \ \sigma(x') = \sigma(c\text{'}arg1_\pi(c_r))$$

Equation 2.68 defines that an execution history X of the sequence of two sequential programs p_1 and p_2 is the concatenation of an execution history X_1 of p_1 and an execution history X_2 of p_2.

Equation 2.69 defines that the execution histories of the if-then-else statement are the execution histories of the if statement when the condition is true and the execution histories of the else statement when the condition is false.

The semantics of basic, sequentially-consistent and linearizable objects are already defined in the previous subsection (Definitions 1, 3 and 2). The semantics of a basic object is the set of execution histories that it allows. The semantics of a sequentially-consistent and linearizable object is the set of pairs of execution and linearization histories that it allows.

Equations 2.70 and 2.71 define the execution histories of a specification. An execution history of $\pi = (\mathcal{P}, \mathcal{D}, \mathcal{T})$ is a history that meets the semantics of the definitions $\mathcal{D}$ and the program $\mathcal{P}$ and also the semantics of the base objects in $\mathcal{T}$. An execution history of the parallel programs is an interleaving of execution histories of the programs. The history of the initialization program precedes the history of the parallel programs. The sub-history for each object complies with the semantics of the object.

Consider an execution triple $(X, \sigma, \mathcal{L})$ in $[\![\pi]\!]$. X is a *symbolic* execution history of π where variables are not yet substituted with their values. σ is the mapping from variables to values at the end of the execution. Applying σ to X, $\sigma(X)$, yields a *concrete* execution history of π. $\mathbb{H}(\pi)$ is the set of concrete execution histories of π. $\mathcal{L}$ is the mapping from objects to their linearization in the execution.

As the focus of the semantics is modeling the concurrency aspects of the specification, it does not model index range of arrays and value range of base objects. These issues are orthogonal to the focus of this semantics and can be studied independently.

Chapter 3

TM Correctness

3.1 Introduction

A transactional memory (TM) is a concurrent object with the four *init*, *read*, *write* and *commit* methods. The clients of a TM are transactions, a sequence of *init* and then *read* and *write* invocations that are possibly succeeded by a *commit* invocation. A transactional processing system is the composition of a TM and as set of clients. The clients issue the invocation events and the TM issues the response events. TM should guarantee that every concurrent execution of an arbitrary set of client transactions is indistinguishable from a sequential execution of them. Correctness conditions for TM such as opacity [28], VWC [44], and TMS1 and TMS2 [22] define the indistinguishably criterion and the set of correct histories. In this chapter, we present a formal definition of opacity.

Design and verification of TM algorithms has been a topic of recent attention and has proved to be formidable. TM algorithms are subtle and prone to bugs. Verification of TM algorithms is a hard in part because the target correctness criterion is a monolithic complicated condition. Can the correctness of TM be stated as a conjunction of simpler meaningful conditions? In other words, is there an intuitive functional decomposition of TM correctness conditions? What are the separate invariants that the TM designers should

maintain? In an early work, Tasiran [74] presented a decomposition of the correctness condition for a specific class of algorithms.

We present intuitive invariants for the correctness of TM algorithms. We say that a history is markable if there is a specific ordering relation called marking such that three invariants are satisfied. These invariants are not only sufficient but also required for opacity. We prove the equivalence of markability and opacity. Roughly speaking, the first invariant called write-observation requires that each read operation returns the most current value and the second invariant called read-preservation requires that the location which is read is not overwritten in a certain interval and the third invariant is the well-known real-time-preservation property.

Separation of concerns brings modularity in understanding, design and verification. Decomposition of the correctness condition informs designers by showcasing different aspects of correctness and helps them concentrate on maintaining one aspect at a time. It also allows studying the effect of separate aspects of correctness on performance. In addition, separation has obvious benefits of modularity and scalability for verification. The marking relation can be defined using the execution order or the linearization order of method calls on the used synchronization objects. Thus, proofs of markability can be aided by and mirror design intuitions. Markability can be proved using the program logic that we will present in the following chapters.

In this section, we first formalize opacity. Then, we introduce the notion of markability and prove the equivalence of opacity and markability.

3.2 Opacity

In this section, we present a formal definition of opacity. Opacity of a TM algorithm is defined in two steps. First, it is defined what it means for a transaction history to be opaque which is called final-state-opacity. Then, a TM algorithm is defined to be opaque if

$$
\begin{aligned}
TReads(H) &= \{R \mid R \in H \ \wedge\ obj_H(R) = this \ \wedge && (3.1)\\
&\qquad name_H(R) = read \ \wedge\ retv_H(R) \neq \mathbb{A}\}\\
TWrites(H) &= \{W \mid W \in H \ \wedge\ obj_H(W) = this \ \wedge && (3.2)\\
&\qquad name_H(W) = write \ \wedge\ retv_H(W) \neq \mathbb{A}\}\\
Commits(H) &= \{C \mid C \in H \ \wedge\ obj_H(C) = this \ \wedge\ name_H(C) = commit\} && (3.3)\\
Trans(H) &= \{T \mid \exists l \in H\colon thread_H(l) = T\} && (3.4)\\
TSequential &= \{S \in THistory \mid \preccurlyeq_S \text{ is a total order of } Trans(S)\} && (3.5)\\
Committed(H) &= \{T \mid \exists l \in Commits(H) \ \wedge\ retv_H(l) = \mathbb{C}\} && (3.6)\\
Aborted(H) &= \{T \mid \exists l \in H\colon obj_H(l) = this \ \wedge\ thread_H(l) = T \ \wedge\ retv_H(l) = \mathbb{A}\} && (3.7)\\
Completed(H) &= Committed(H) \cup Aborted(H) && (3.8)\\
Live(H) &= Trans(H) \setminus Completed(H) && (3.9)\\
TComplete &= \{H \in THistory \mid \forall T \in Trans(H)\colon T \in Completed(H)\} && (3.10)\\
CommitPending(H) &= \{T \in Live(H) \mid \exists l \in H\colon && (3.11)\\
&\qquad thread_H(l) = T \ \wedge\ name_H(l) = commit\\
&\qquad iEv(l) \Subset H \ \wedge\ \neg(rEv(l) \Subset H)\}\\
TExtension(H) &= \{H' \in THistory \mid \exists H''\colon H' = H \cdot H'' && (3.12)\\
&\quad \forall T \in Trans(H') \Rightarrow T \in Trans(H) \ \wedge\\
&\quad Live(H) \setminus CommitPending(H) \subseteq Aborted(H') \ \wedge\\
&\quad CommitPending(H) \subseteq Completed(H')\}\\
Visible(S,T) &= filter\,(S, \lambda T'.(T' = T) \vee ((T' \prec\!\!\!\prec_S T) \wedge T' \in Committed(S))) && (3.13)\\
NoWriteBetween_S(W,R) &= \forall W' \in TWrites(S)\colon W' \preceq_S W \ \vee\ R \prec_S W' && (3.14)\\
SeqSpec(i) &= \{S \in Sequential \mid \forall R \in TReads(S)\colon \exists W \in TWrites(S)\colon && (3.15)\\
&\qquad W \prec_S R \ \wedge\ NoWriteBetween_S(W,R) \ \wedge\\
&\qquad retv_S(R) = arg2_S(W)\}\\
TSeqSpec &= \{S \in TSequential \cap TComplete \mid \forall T \in S\colon \forall i \in I\colon && (3.16)\\
&\qquad (Visible(S,T) \mid i) \in SeqSpec(i)\}\\
FinalStateOpaque &= \{H \in THistory \mid \exists H' \in TExtension(H)\colon \exists S \in TSequential\colon && (3.17)\\
&\qquad H' \equiv S \wedge\ \preccurlyeq_{H'} \subseteq \preccurlyeq_S \ \wedge S \in TSeqSpec\}
\end{aligned}
$$

Figure 3.1: *FinalStateOpaque*

every transaction history of every source program running on top of that TM algorithm is final-state-opaque.

A *transaction history* H is an execution history such that $H|mem = H_{Init} \cdot H'$ with the following conditions. H_{Init} is the following history that initializes every location to v_0. $H_{Init} = l_{0i} \triangleright init_{T_0}() \cdot l_{00} \triangleright write_{T_0}(1, v_0){:}ok \cdot \ldots \cdot l_{0m} \triangleright write_{T_0}(m, v_0){:}ok \cdot l_{0c} \triangleright commit_{T_0}{:}\mathbb{C}$. For every $T \in H'$, the history $H'|T$ is a prefix of $e.e'$. The event sequence e is the initialization method call $l \triangleright init_T()$ (for some l), and then a sequence of reads $l \triangleright read_T(i){:}v$ and writes $l \triangleright write_T(i, v)$ (for some l, i, and v). The event sequence e' is one of the following sequences (for some l, i, and v): (1) $inv(l \triangleright read_T(i))$, $ret(l \triangleright \mathbb{A})$, (2) $inv(l \triangleright write_T(i, v))$, $ret(l \triangleright \mathbb{A})$, (3) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{C})$, (4) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{A})$, or (5) $inv(l \triangleright abort_T())$, $ret(l \triangleright \mathbb{A})$. Let $THistory$ denote the set of transaction histories. Let $Trans(H)$ denote the set of transactions of H. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i. For a TM algorithm specification π, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that π results.

$FinalStateOpaque$ is defined in Figure 3.1. First, we present some preliminary definitions. We use T prefix before some of the terms for transactions to avoid confusion with the terms for concurrent objects. We say that a transaction history is *transaction sequential* if it is a sequence of transactions. A transaction T is *committed* or *aborted* in a transaction history H if there is respectively a commit or abort response event for T in H. A *completed* transaction is either committed or aborted. A *live* transaction is a transaction that is not completed. A transaction history is *complete* if all its transactions are completed. A *pending* transaction has a pending event and a *commit-pending* transaction has a commit pending event. An *extension* of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions.

If H is a transaction history and p is a predicate on transaction identifiers, we define $filter(H, p)$ to be the subsequence of H that contains the events of transactions T for which $p(T)$ is true. The *visible history* for a transaction T in a sequential transaction history S,

$Visible(S, T)$, is the sequence of committed transactions before T in S and T itself. The *sequential specification* of a location i, $SeqSpec(i)$, is the set of sequential histories of read and write method calls on location i where every read returns the value given as the argument to the latest preceding write (regardless of transaction identifiers). It is essentially the sequential specification of a register. *Transactional sequential specification* is the set of complete sequential transaction histories S that for every transaction T and location i, $Visible(S, T)|i$ is a member of the sequential specification of i. A transaction history H is *final-state-opaque* if there is an equivalent sequential transaction history S for an extension of H such that S is real-time-preserving and a member of transactional sequential specification. The sequential history S is called the justifying history. In other words, every correct concurrent execution is indistinguishable from a correct sequential execution.

Definition 14 (Opaque TM Algorithm)**.** *A TM algorithm is opaque if and only if every execution history of it is final-state-opaque.*
$Opaque = \{\pi \mid \mathbb{H}(\pi) \subseteq FinalStateOpaque\}$

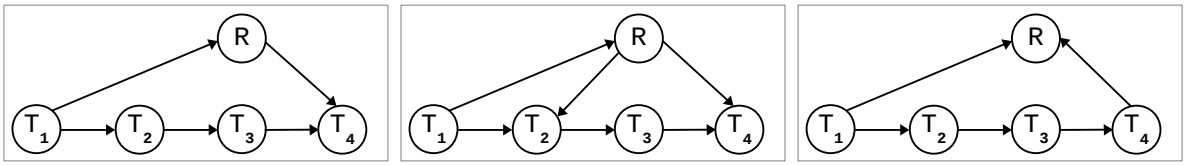
3.3 Markability

3.3.1 Write-observation and Read-preservation

In this section, we explain the main ideas behind markability by focusing on complete histories with only global reads and writes. A history is complete if every transaction in it is either aborted or committed. A read R by a transaction T is global if T has no write to the same location before R. A write W by a transaction T is global if T has no write to the same location after W.

A transaction history is *markable* if and only if there exists a *marking* of it that is *write-observant*, *read-preserving*, and *real-time-preserving*. We explain each term in turn.

A marking of a transaction history is a relation on the union of the *transactions* and the *read operations* in the history. We can think of the marking as the union of a collection of



(a) (b) (c)

Figure 3.2: Illustrations of Write-observation and Read-preservation

orders:

- The *effect order*: The effect order is a total order of the transactions.
- The *access orders*: Consider an unaborted read operation R that reads from a location i. Let writers of i be the committed transactions that have write operation(s) to location i. For each such R, the access order is an antisymmetric relation that orders R and every writer of i.

The effect order represents the order in which the transactions appear to take effect. The access order represents where the read operation's access to the location has happened between the accesses by the writers of that location.

Note that marking not only recognizes the points where transactions take effect but also the points where reads take place. The effect point of a transaction captures the point where the whole transaction takes effect. But a transaction is split into multiple operations. Particularly, read operations observe values before the commit is even invoked. Any value that the TM algorithm returns in response to a read invocation should be justified at the point where the transaction takes effect. There is a point where each writer transaction writes the new value to the shared objects. Every read operation reads the value that it returns at a certain point between the write points of the writer transactions. The access order makes this design decision explicit. The access order makes it possible to decompose the consistency condition into two orthogonal invariants. Particularly, the read-preservation invariant makes sure that the read value is not overwritten in the interval between the point where a read happens and the point where the transaction takes effect. Next, we will explain write-observation and read-preservation.

At a high level, write-observation means that each read operation should read the most current value. Let us explain this idea in more detail. Consider an unaborted read operation R from the location i. Let *pre-accessors* be the writers of i that come before R in the access order for R. We can use the effect order to determine the *last* pre-accessor that is, the pre-accessor that is greatest in the effect order. Write-observation requires that the value that R reads be the same as the value written by the last pre-accessor.

Figure 3.2 illustrates the write-observation and read-preservation invariants. Each sub-figure shows a marking relation $\sqsubseteq$. In every sub-figure, the effect order is $T_1 \sqsubseteq T_2 \sqsubseteq T_3 \sqsubseteq T_4$ and the transaction T_3 performs the read operation R. In Figure 3.2(a), T_1 and $T4$ are writers of i and the access order is $\{T_1 \sqsubseteq R, R \sqsubseteq T_4\}$. T_1 is the last pre-accessor for R. Thus, by write-observation, R is expected to return the value that T_1 writes to i.

At a high level, read-preservation means that the location read by a read operation is not overwritten between the points that the read takes place and the transaction takes effect. Let us explain this idea in more detail. Consider an unaborted read operation R by transaction T from the location i. Intuitively, read-preservation requires that no writer of i comes between R and T in the marking relation. More precisely, read-preservation requires that there is no writer T' of i that accesses i *after* R and takes effect *before* T and there is no writer T' of i that takes effect *after* T and accesses i *before* R. (Note that depending on whether a transaction takes effect earlier or later in its lifetime, one of these two conditions is usually trivially true.) In other words, read-preservation requires the writers to both access i and take effect on the same "side" of R and T. More precisely, if a writer T' accesses i *before* R (T' is marked before R in the access order), then T' takes effect *before* T (T' is marked before T in the effect order) too. Similarly, read-preservation requires that if T' accesses i *after* R, it takes effect *after* T too.

The marking relation in Figure 3.2(a) satisfies read-preservation as there is no writer between R and T_2. The transaction T_1 accesses i before R and takes effect before T_3 too. The transaction T_4 accesses i after R and takes effect after T_3 too. Figures 3.2(b) and 3.2(c)

show markings that are not read-preserving. In Figure 3.2(b), T_1, T_2 and $T4$ are writers of i and the access order is $\{T_1 \sqsubseteq R, R \sqsubseteq T_2, R \sqsubseteq T_4\}$. The transaction T_2 is between R and T_3. Therefore, the marking is not read-preserving. In Figure 3.2(c), T_1 and $T4$ are writers of i and the access order is $\{T_1 \sqsubseteq R, T_4 \sqsubseteq R\}$. The transaction $T4$ is between T_3 and R. Therefore, the marking is not read-preserving.

The real-time-preservation condition requires that if all the events of a transaction T happen before all the events of another transaction T', then T is less than T' in the effect order.

Our marking theorem says that a history is opaque if and only if it is markable. So, to prove opacity, we can focus on proving markability. The algorithm designer can usually define the marking relation readily from the guarantees (such as linearization orders) of the used shared objects.

If a transaction history H is markable, we can show that H is opaque. We construct a justifying history by ordering the transactions in the effect order. Consider an arbitrary read R from i by T. We call the writers of i that take effect before T, pre-effectors. Let the last pre-effector be the pre-effector that is the greatest in the effect order. We need to show that the value that R returns is the value that the last pre-effector writes. We remind that we call the writers that access i before R, pre-accessors. First, we argue that pre-accessors are exactly pre-effectors. If a writer accesses before R, by read-preservation, it does not take effect after T. Thus, by totality of effect order, it takes effect before T. If a writer takes effect before T, by read-preservation, it does not access after R. Thus, as the access order orders R and every writer of i, T accesses before R. Second, from write-observation, we have that R returns the value written by the last pre-accessor in the effect order. Thus from the two above statements, we have that R returns the value written by the last pre-effector (in the effect order). This is the essence of the condition needed to prove opacity.

T		T'	
$\boxed{I01 \triangleright}$	$snap = clock.read()$	$C02_i \triangleright$	$lock[i].trylock()$
$I02 \triangleright$	$rver[t].write(snap)$		...
		$\boxed{C07 \triangleright}$	$wver = clock.iaf()$
	...		...
		$\boxed{C16_i \triangleright}$	$reg[i].write(v)$
$\boxed{R04 \triangleright}$	$v = reg[i].read()$	$C17_i \triangleright$	$ver[i].write(wver)$
$R05 \triangleright$	$l = lock[i].read()$	$C18_i \triangleright$	$lock[i].unlock()$
$R06 \triangleright$	$s_2 = ver[i].read()$		
$R07 \triangleright$	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 3.3: TL2 Read-Preservation Example

3.3.2 Marking TL2

Let us look at marking of TL2 algorithm [18] as an example. TL2 is specified in Figure 2.2.

Let us describe the marking relation for TL2. The *clock* object numbers the snapshots. Every transaction reads an initial snapshot number at $I01$. A committing transaction makes a new snapshot at $C07$. The effect point of a TL2 transaction is $I01$, if it is live or aborted and, is $C07$, if it is committed. The effect order of transactions is the linearization order of *clock* for their effect points. The access point of a read operation is at $R04$ where $reg[i]$ is read and the access point of a writer of i is $C16_i$ where $reg[i]$ is written. Consider a read R from i and a writers T' to i. If the access point of T' is executed before the access point of R, then T' is ordered before R in the access order of R. Otherwise, T' is ordered after R in the access order of R.

One of the two conjuncts of the read-preservation property requires that for every transaction T with an unaborted read operation R from a location i, there is no writer T' of i such that T' takes effect after T and accesses i before R. Let us see how TL2 preserves this property. We assume that there exists such a writer T' and show that the validation checks embodied in TL2 detect the existence of T' and abort R. We consider a transaction T with

a read operation R from a location i and a writer T' of i. We assume that T' takes effect after T and T' accesses i before R. For brevity, we consider only the case that T is a live or aborted (not a committed) transaction. Figure 3.3 depicts the two transactions. We use the binary operators $\prec_X$ to denote execution precedence, $\sim_X$ to denote concurrent execution and $\precsim_X$ to denote precedence or concurrent execution of method calls. We use the binary operators $\prec_{clock}$, $\prec_{ver[i]}$ and $\prec_{lock[i]}$ to denote the linearization order of $clock$, $ver[i]$ and $lock[i]$ respectively.[1] We remind that the real-time-preservation property of a linearizable object o states that if a method call m_1 on o is executed before another method call m_2 on o, then m_1 is linearized before m_2. Equivalently, if m_1 is linearized before m_2, then m_1 is executed before or concurrent to m_2. By the marking relation defined above, from the premise that T' takes effect after T, we have (1) $I01 \prec_{clock} C07$ and from the premise that T' accesses i before R, we have (2) $C16_i \prec_{reg[i]} R04$. The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and show that R returns $\mathbb{A}$ in both cases.

- Case 1: (3) $R05 \prec_{lock[i]} C18_i$. From the execution, we have (4) $C02_i \prec_X C16_i$ and (5) $R04 \prec_X R05$. By the real-time-preservation property for $ver[i]$ on 2, we have (6) $C16_i \precsim_X R04$. By the transitivity of the execution order on 4, 6 and 5, we have $C02_i \prec_X R05$; thus, by the real-time-preservation property for $lock[i]$, we have (7) $C02_i \prec_{lock[i]} R05$. From 7 and 3, we have that $R05$ is executed when $lock[i]$ is acquired. Therefore, $R05$ returns $true$ i.e. $l = true$. Thus, the validation check fails and R returns $\mathbb{A}$.
- Case 2: (8) $C18_i \prec_{lock[i]} R05$. By the real-time-preservation property for $lock[i]$, from 8, we have (9) $C18_i \precsim_X R05$. From the execution, we have (10) $C17_i \prec_X C18_i$ and (11) $R05 \prec_X R06$. By the transitivity of the execution order on 10, 9 and 11, we have (12) $C17_i \prec_X R06$. By the real-time-preservation property for $ver[i]$, from 12, we have (13)

[1] We have formally proved the markability of TL2 using a novel program logic [**?**] that facilitates reasoning about execution and linearization orders. To keep the focus of this paper on markability, we present a simplified reasoning instead of the formal presentation of the logic.

$$
\begin{array}{rcl}
Committed(H) & = & \{T \mid \exists l \in H\colon obj_H(l) = mem \ \wedge \ trans_H(l) = T \ \wedge \ retv_H(l) = \mathbb{C}\} \\
Aborted(H) & = & \{T \mid \exists l \in H\colon obj_H(l) = mem \ \wedge \ trans_H(l) = T \ \wedge \ retv_H(l) = \mathbb{A}\} \\
Completed(H) & = & Committed(H) \cup Aborted(H) \\
Live(H) & = & Trans(H) \setminus Completed(H) \\
CommitPending(H) & = & \{T \in Live(H) \mid \exists l \in H\colon trans_H(l) = T \ \wedge \ name_H(l) = commit \\
 & & \quad iEv(l) \Subset H \ \wedge \ \neg(rEv(l) \Subset H)\} \\
TExtension(H) & = & \{H' \in THistory \mid \exists H''\colon H' = H \cdot H'' \\
 & & \quad \forall T \in Trans(H') \Rightarrow T \in Trans(H) \wedge \\
 & & \quad Live(H) \setminus CommitPending(H) \subseteq Aborted(H') \wedge \\
 & & \quad CommitPending(H) \subseteq Completed(H')\} \\
TReads(H) & = & \{R \mid R \in H \ \wedge \ obj_H(R) = mem \ \wedge \ name_H(R) = read \ \wedge \ retv_H(R) \neq \mathbb{A}\} \\
TWrites(H) & = & \{W \mid W \in H \ \wedge \ obj_H(W) = mem \ \wedge \ name_H(W) = write \ \wedge \ retv_H(W) \neq \mathbb{A}\} \\
LocalTReads(H) & = & \{R \mid R \in TReads(H) \ \wedge \ \exists W \in TWrites(H)\colon \\
 & & \quad trans_H(R) = trans_H(W) \ \wedge \ arg1_H(R) = arg1_H(W) \ \wedge \ W \prec_H R\} \\
GlobalTReads(H) & = & TReads(H) \setminus LocalTReads(H) \\
LocalTWrites(H) & = & \{W \mid W \in TWrites(H) \ \wedge \ \exists W' \in TWrites(H)\colon \\
 & & \quad trans_H(W) = trans_H(W') \ \wedge \ arg1_H(W) = arg1_H(W') \ \wedge \ W \prec_H W'\} \\
GlobalTWrites(H) & = & TWrites(H) \setminus LocalTWrites(H) \\
Writers_H(i) & = & \{T \in Trans(H) \mid \exists l \in TWrites(H)\colon arg1_H(l) = i \wedge \\
 & & \quad trans_H(l) = T \ \wedge \ T \in Committed(H)\}
\end{array}
$$

Figure 3.4: The set of local and global reads and writes

$C17_i \prec_{ver[i]} R06$. It is straightforward to separately prove that (14) The register $ver[i]$ is updated only to ascending numbers. From 14 and 13, we have that $R06$ reads a value that is greater than or equal to the value that $C17_i$ writes i.e. (15) $s_2 \geq wver$. From 1, and that *iaf* returns the incremented value, we have (16) $snap < wver$. The value of *sver* is read at $R07$ from *rver*. The thread-local register *rver* is only assigned at $I02$ to *snap*. Thus, we have (17) $snap = sver$. From 15, 16 and 17, we have $s_2 > sver$. Thus, the validation check fails and R returns $\mathbb{A}$ in this case too.

Please see section 10.2.2 for more details about the proof of markability of TL2.

$$
\begin{aligned}
&Marking(H) = \{\sqsubseteq\ \mid \\
&\qquad \forall T1, T2, T3 \in Trans(H): \\
&\qquad\quad (T1 \sqsubseteq T2\ \vee\ T2 \sqsubseteq T1) \wedge \\
&\qquad\quad (T1 \sqsubseteq T2\ \wedge\ T2 \sqsubseteq T1) \Rightarrow (T1 = T2) \wedge \\
&\qquad\quad (T1 \sqsubseteq T2)\ \wedge\ (T2 \sqsubseteq T3) \Rightarrow (T1 \sqsubseteq T3) \wedge \\
&\qquad \forall R, T: \mathit{Let}\ i = arg1_H(R): (R \in GlobalTRead(H)\ \wedge\ T \in Writers_H(i)) \Rightarrow \\
&\qquad\quad (R \sqsubseteq T\ \vee\ T \sqsubseteq R) \wedge \\
&\qquad\quad (R \sqsubseteq T \Rightarrow \neg T \sqsubseteq R)\ \wedge\ (T \sqsubseteq R \Rightarrow \neg R \sqsubseteq T)\} \\
&NoWriteBetween_H(W, R) \Leftrightarrow \\
&\qquad \forall W' \in TWrites(H): W' \preceq_H W\ \vee\ R \prec_H W' \\
&LocalWriteObs(H) \Leftrightarrow \\
&\qquad \forall R \in LocalTReads(H): \mathit{Let}\ T = trans_H(R), i = arg1_H(R), H' = H|T|i: \\
&\qquad \exists W \in TWrites(H'): W \prec_{H'} R\ \wedge\ NoWriteBetween_{H'}(W, R)\ \wedge\ retv_{H'}(R) = arg2_{H'}(W) \\
&NoWriterBetween_{H,i}(x, \sqsubseteq, x') \Leftrightarrow \\
&\qquad \forall T \in Writers_H(i): T \sqsubseteq x\ \vee\ x' \sqsubseteq T \\
&LastPreAccessor_{H,\sqsubseteq}(T', R) \Leftrightarrow \mathit{Let}\ i = arg1_H(R), T = trans_H(R): \\
&\qquad T' \in Writers_H(i)\ \wedge\ T' \neq T\ \wedge\ T' \sqsubset R\ \wedge\ NoWriterBetween_{H,i}(T', \sqsubseteq, R) \\
&GlobalWriteObs(H, \sqsubseteq) \Leftrightarrow \\
&\qquad \forall R \in GlobalTReads(H): \exists W \in GlobalTWrites(H): \mathit{Let}\ T' = trans_H(W): \\
&\qquad LastPreAccessor_{H,\sqsubseteq}(T', R)\ \wedge\ arg1_H(R) = arg1_H(W)\ \wedge\ retv_H(R) = arg2_H(W) \\
&WriteObs(H, \sqsubseteq) \Leftrightarrow \\
&\qquad LocalWriteObs(H)\ \wedge\ GlobalWriteObs(H, \sqsubseteq) \\
&ReadPres(H, \sqsubseteq) \Leftrightarrow \\
&\qquad \forall R \in GlobalTReads(H): \mathit{Let}\ i = arg1_H(R), T = trans_H(R): \\
&\qquad NoWriterBetween_{H,i}(R, \sqsubseteq, T)\ \wedge\ NoWriterBetween_{H,i}(T, \sqsubseteq, R) \\
&RealTimePres(H, \sqsubseteq) \Leftrightarrow \\
&\qquad \preccurlyeq_H\ \subseteq\ \sqsubseteq \\
&FinalStateMarkable = \{H \in THistory \mid \exists H' \in TExtension(H): \exists \sqsubseteq\ \in Marking(H'): \\
&\qquad\qquad ReadPres(H', \sqsubseteq)\ \wedge\ WriteObs(H', \sqsubseteq)\ \wedge\ RealTimePres(H', \sqsubseteq)\}
\end{aligned}
$$

Figure 3.5: *FinalStateMarkable*

3.3.3 The Marking Theorem

In this section, we define markability for general histories and present the marking theorem that states the equivalence of opacity and markability.

First, we present some preliminary definitions in Figure 3.4. (We use the prefix T before some of the terms for transactions to avoid confusion with similar terms that are usually used for general concurrent objects.) A transaction T is *committed* or *aborted* in a transaction history H if there is respectively a commit or abort response event for T in H. A *completed* transaction is either committed or aborted. A *live* transaction is a transaction that is not completed. A *pending* transaction has a pending event and a *commit-pending* transaction has a commit pending event. An *extension* of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions.

A *local read* is a read that is preceded by a write by the same transaction to the same location. Intuitively, a local read should read a value that is previously written by the same transaction and hence the name. A *global read* is a read that is not local. A *local write* is a write that precedes a write by the same transaction to the same location. A local write is overwritten by the same transaction and hence the name. A *global write* is a write that is not local. The *writers of* i are the committed transactions that write to location i.

Markability is defined in Figure 3.5. A *marking* $\sqsubseteq$ of a transaction history is the union of the following relations on the set of transactions and the global reads.

- The *effect order*: The set of transactions is totally ordered by the marking relation $\sqsubseteq$. In other words, the marking relation $\sqsubseteq$ is total, antisymmetric and transitive on the set of transactions.
- The *access orders*: For each global read R from a location i, R and every writer of i are ordered by the marking relation $\sqsubseteq$. In other words, the marking relation $\sqsubseteq$ totally orders every global read R from a location i with respect to writers of i and is antisymmetric.

The *write-observation* property is comprised of the two properties: *local write-observation* and *global write-observation*. Local write-observation requires that every local read R from a location i returns the value written by the last write to i that is executed before R by the same transaction. We remind that pre-accessors of R are the writers of i that are ordered before R in the access order and the last pre-accessor is the one that is greatest in the effect order. Global write-observation requires that the value that every global read R from a location i returns is the value written by the global write to i by the last pre-accessor transaction of R.

The *Read-preservation* property requires that for every global read R from location i by transaction T, there is no writer transaction T' of i such that T' is marked between R and T (i.e. T' accesses i after R and takes effect before T), or similarly, T' is marked between T and R (i.e. T' takes effect after T and accesses i before R).

The *real-time-preservation* property requires that if T is before T' in the real-time order, then T takes effect before T' as well.

A transaction history is *final-state-markable* if and only if there exists a marking for an extension of it that is write-observant, read-preserving, and real-time-preserving.

The marking theorem states that a transaction history is final-state-opaque if and only if it is final-state-markable.

Theorem 1. ***(Marking)*** $FinalStateOpaque = FinalStateMarkable.$

Please see the appendix section 10.2.1 for the proofs.

Chapter 4

Testing TM Algorithms

4.1 Introduction

Considering the correctness conditions of transactional memory, designing correct transactional memory algorithms is a formidable task. Algorithm design is an iterative process of trying alternatives, fixing issues and improving the performance. An update to the algorithm makes the algorithm work for a specific new scenario but should preserve the correctness of the algorithm on the existing scenarios as well. A tool that tests for specific scenarios can assist algorithm designers during both the design and the maintenance of the algorithm. The tool can be used for regression testing of known scenarios. In this chapter, we identify specific pitfalls that lead to non-opacity and show how a tool can automatically find such pitfalls in the algorithms.

We identify two problems that lead to non-opacity: the write-skew anomaly and the write-exposure anomaly. The write-skew anomaly is an incorrectness pattern that is known in the setting of databases [6]. The write-exposure anomaly happens when a TM algorithm exposes written values to other transactions before the transaction commits.

We present a tool called Samand that automatically finds such problems. The tool inputs a TM algorithm, a program and a test assertion. The test assertion can be a partial

correctness condition such as negation of a bug pattern. If an execution of the test program can violate the test assertion, Samand outputs a violating trace. Samand translates the specification into constraints and feeds them to Z3 SMT solver [17]. If the constraints are satisfiable, the tool reconstructs and outputs a violating program trace.

We show that the well-known TM algorithms DSTM and McRT don't satisfy opacity. DSTM suffers from the write-skew anomaly, while McRT suffers from the write-exposure anomaly. These results may be surprising because previous work has proved that DSTM and McRT satisfy opacity [31, 30]. However, there is no conflict and no mystery: the previous work focused on abstractions of DSTM and McRT, while we work with specifications that are much closer to original formulations of DSTM and McRT. Thus, we experience a common phenomenon: once we refine a specification, we may lose some properties. We present fixes to both DSTM and McRT that we conjecture make the fixed algorithms satisfy opacity.

In the following subsections, we first introduce bug patterns that violate opacity. Then, we introduce our tool and the set of constraints that it generates. Finally, we present the result of applying our tool to DSTM and McRT algorithms.

4.2 Opacity Bug Patterns

Consider the following transaction histories:

$$
\begin{aligned}
H_{WS} &= Init \cdot read_{T_1}(1){:}v_0 \cdot read_{T_2}(1){:}v_0 \cdot read_{T_1}(2){:}v_0 \cdot read_{T_2}(2){:}v_0 \cdot \\
&\quad write_{T_1}(1, -v_0) \cdot write_{T_2}(2, -v_0) \cdot \\
&\quad inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{C}) \cdot ret_{T_2}(\mathbb{C}) \\
H_{WE} &= Init \cdot inv_{T_1}(read_{T_1}(2)) \cdot write_{T_2}(2, v_1) \cdot ret_{T_1}(v_1) \cdot \\
&\quad inv_{T_2}(read_{T_2}(1)) \cdot write_{T_1}(1, v_1) \cdot ret_{T_2}(v_1) \cdot \\
&\quad inv_{T_1}(commit_{T_1}) \cdot inv_{T_2}(commit_{T_2}) \cdot ret_{T_1}(\mathbb{A}) \cdot ret_{T_2}(\mathbb{A}) \\
H_{WE2} &= Init \cdot inv_{T_1}((write_{T_1}(1, v_1)) \cdot read_{T_2}(1){:}v_1 \cdot ret_{T_1}(ok) \cdot \\
&\quad write_{T_1}(1, v_2) \cdot commit_{T_1}(){:}\mathbb{C} \cdot commit_{T_2}(){:}\mathbb{A}
\end{aligned}
$$

where $Init$ is described below and H_0 is a transaction history that does not contain a write operation that writes value j.

In Appendix, we prove that none of the transaction histories H_{WS}, H_{WE}, H_{WE2} are opaque.

Theorem 2. $\{H_{WS}, H_{WE}, H_{WE2}\} \cap FinalStateOpaque = \emptyset$.

We say that H_{WS}, H_{WE} are *bug patterns*, because if a TM can produce any of them, then the TM violates opacity. Let us now focus on H_{WS}, H_{WE} and later turn to H_{WE2}.

Write-skew anomaly. The transaction history H_{WS} is evidence of the write-skew anomaly. Let us illustrate the write-skew anomaly with the following narrative.

Assume that a person has two bank accounts that are stored at locations i_1 and i_2 and that have the initial balances v_0 and v_0, where $v_0 > 0$. Assume also that the regulations of the bank require the *sum* of a person's accounts to be positive or zero. Thus, the bank will authorize a transaction that withdraws the sum of the two accounts from one of them i.e.

updates the value of one of the accounts with the previous value of the account minus the sum of the two accounts.

Now we interpret the narrative in the context of H_{WS}, which is a record of the execution of two "bank-authorized" transactions. In H_{WS} the transaction T_1 reads the values of both accounts and updates i_1 with $v_0 - (v_0 + v_0) = -v_0$. Similarly, the transaction T_2 reads the values of both accounts and updates i_2 with $-v_0$. But in H_{WS} both transactions commit, which results in a state that violates the regulations of the bank: $-v_0$ is the balance of both accounts.

The problem with H_{WS} stems from that the TM that produced H_{WS} doesn't guarantee noninterleaving semantics of the transactions. In a noninterleaving semantics, either T_1 appears to execute before T_2, or T_2 appears to execute before T_1. However, if we order T_1 before T_2, then the values read by T_2 violate correctness; and if we order T_2 before T_1, then the values read by T_1 violate correctness.

The reader may notice that since H_{WS} is not opaque and all the transactions in H_{WS} are committed, H_{WS} is not even serializable. However, H_{WS} does satisfy *snapshot isolation*, which is a necessary, though not a sufficient, condition for serializability. A history satisfies snapshot isolation if its reads observe a consistent snapshot. Snapshot isolation prevents observing some of the updates of a committing transaction before the commit and some of the rest of the updates after the commit. Algorithms that support only snapshot isolation are known to be prone to the write-skew anomaly, as shown by Berenson et al. [6].

Write-exposure anomaly. The transaction history H_{WE} is evidence of the write-exposure anomaly. The two locations i_1 and i_2 each has initial value v_0 and no *committed* transaction writes a different value to them, and yet the two read operations return the value v_1. Write-exposure happens when a transaction that eventually fails to commit *writes* to a location i and *exposes* the written value to other transactions that read from i. Thus, active or aborting transactions can read inconsistent values. This violates opacity even if these

transactions are eventually prevented from committing.

4.3 Automatic Bug Finding

We present a tool called Samand in which inputs a specification consisting of a TM algorithm, a user program, and a test assertion. A input specification is *correct* if every execution of the user program satisfies the test assertion. Our tool solves constraints to decide whether a input specification is correct.

Our language. We present Samand via two examples. We will use a sugared notation, for simplicity. We explain the actual language and code for both examples in the appendix 10.3. The first example is

$$(\pi_{DSTM}, P_{WS}, \neg WS)$$

where π_{DSTM} (see Figure 2.4) is a core version of the TM algorithm DSTM, and the user program and the test assertion are:

$$\begin{aligned} P_{WS} &= \{r_{11} = read_{T_1}(1);\ \ r_{12} = read_{T_1}(2);\ \ write_{T_1}(1, v_1);\ \ c_1 = commit_{T_1}()\} \ || \\ &\quad\ \{r_{21} = read_{T_2}(1);\ \ r_{22} = read_{T_2}(2);\ \ write_{T_2}(2, v_1);\ \ c_2 = commit_{T_2}()\} \\ WS &= (r_{11} = v_0 \ \wedge\ r_{12} = v_0 \ \wedge\ r_{21} = v_0 \ \wedge\ r_{22} = v_0 \ \wedge\ c_1 = \mathbb{C} \ \wedge\ c_2 = \mathbb{C}) \end{aligned}$$

Note that the assertion WS specifies a *set* of buggy histories of the user program; the history H_{WS} is a member of that set.

The second example is

$$(\pi_{McRT}, P_{WE}, \neg WE)$$

where π_{McRT} (see Figure 2.6) is a core version of the TM algorithm McRT, and the user

program and the test assertion are:

$$\begin{aligned} P_{WE} &= \{r_1 = read_{T_1}(2);\ \ write_{T_1}(1, v_1);\ \ c_1 = commit_{T_1}()\} \ || \\ &\quad\ \{write_{T_2}(2, v_1);\ \ r_2 = read_{T_2}(1);\ \ c_2 = commit_{T_2}()\} \\ WE &= (r_1 = v_1 \ \wedge\ r_2 = v_1 \ \wedge\ c_1 = \mathbb{A} \ \wedge\ c_2 = \mathbb{A}) \end{aligned}$$

Similar to the first example, the assertion WE specifies a *set* of buggy histories of the user program; the history H_{WE} is a member of that set.

Samand enables specification of loop-free user programs. Every user program has a finite number of possible executions and those executions all terminate.

Constraints. Samand uses the following notion of constraints to decide whether the input specification is correct. Let l, x, v range over finite sets of labels, variables, and values, respectively. A constraint is an assertion about histories X and is generated by the following grammar:

a	$::=$	$obj_X(l) = o \ \mid\ name_X(l) = n \ \mid\ thread_X(l) = T \ \mid$	Assertion
		$arg1_X(l) = u \ \mid\ arg2_X(l) = u \ \mid\ retv_X(l) = x \ \mid$	
		$l \in X \ \mid\ l \prec_X l \ \mid\ e_1 \lhd_X e_2$	
		$\neg a \ \mid\ a \ \wedge\ a$	
e	$::=$	$iEv(l) \ \mid\ rEv(l)$	Event
u	$:=$	$v \ \mid\ x$	Variable or Value

The events $iEv(l)$ and $rEv(l)$ are the invocation and response events of l respectively. The assertions $obj_X(l) = o$, $name_X(l) = n$, $thread_X(l) = T$, $arg1_X(l) = u$, $arg2_X(l) = u$, and $retv_X(l) = x$ respectively assert that the receiver object of l is o, the method name of l is n, the calling thread of l is T, the first argument of l is u, the second argument of l is u, and the return value of l is x. The assertion $l \in X$ asserts that l is in the history X. The assertion $l \prec_X l'$ asserts that l is executed before l'. The assertion $e_1 \lhd_X e_2$ asserts that the

event e_1 is before the event e_2 in the history X. The satisfiability problem is to decide, for a given constraint, whether there exists histories that satisfy the constraint.

From programs to constraints. Samand maps the input specification to a set of constraints such that the input specification is correct if and only if the constraints are unsatisfiable.

We defined run-time labels and labeled variables in section 2.3.3. A label c_1'c_2 denotes a method call annotated with c_2 that is executed inside the body of a *this* method call annotated with c_1. On the other hand, the label c_1 denotes a *this* method call in the top-level statements of the program annotated with c_1. For each runtime label l, we consider the pair of invocation event $iEv(l)$ and response event $rEv(l)$. The variable c'x denotes the local variable x insider a *this* method call labeled c.

The relation $\lhd_X$ is a total order on events. We assert that the $\lhd_X$ is an anti-symmetric, transitive and total relation on events. For each pair of events e_1 and e_2, we generate the following constraints.

$$e_1 \lhd_X e_2 \Rightarrow \neg(e_2 \lhd_X e_1) \tag{4.1}$$

$$(e_1 \lhd_X e_2) \ \wedge \ (e_2 \lhd_X e_3) \Rightarrow (e_1 \lhd_X e_3) \tag{4.2}$$

$$(e_1 \lhd_X e_2) \ \vee \ (e_2 \lhd_X e_1) \tag{4.3}$$

The invocation event is before its response event. For every label l, we generate the following constraint.

$$iEv(l) \lhd_X rEv(l) \tag{4.4}$$

Every method call inside a *this* method call is before the invocation and after the response event of the *this* method call. Consider a *this* method call labeled c that calls the method n.

Let $\overline{c_i}$ be the set of labels of the body of n. For each c_i, we generate the following constraint.

$$iEv(c) \lhd_X iEv(c\text{'}c_i) \ \wedge \ rEv(c\text{'}c_i) \lhd_X rEv(c) \tag{4.5}$$

A method call is executed before another method call if the response event of the former is before the invocation event of the latter.

$$l_1 \prec_X l_2 \Leftrightarrow rEv(l_1) \lhd_X iEv(l_2) \tag{4.6}$$

$$l_1 \preceq_X l_2 \Leftrightarrow (l_1 \prec_X l_2 \ \vee \ l_1 = l_2) \tag{4.7}$$

For each method call, we generate constraints that assert the components of the method call. For a *this* method call

$$c \rhd \ x = n_\tau(u)$$

we generate the following constraints.

$$\begin{aligned} &obj_X(c) = this \ \wedge \ thread_X(c) = \tau \ \wedge \\ &name_X(c) = n \ \wedge \ arg1_X(c) = u \ \wedge \\ &retv_X(c) = x \end{aligned} \tag{4.8}$$

For every method call

$$c_i \rhd \ x = \phi[i].n_\tau(u)$$

inside a *this* method call labeled c, we generate the following constraints.

$$\begin{aligned} &obj_X(c\text{'}c_i) = \phi[c\text{'}i] \ \wedge \ thread_X(c\text{'}c_i) = c\text{'}\tau \ \wedge \\ &name_X(c\text{'}c_i) = n \ \wedge \ arg1_X(c\text{'}c_i) = c\text{'}u \ \wedge \\ &retv_X(c\text{'}c_i) = c\text{'}x \end{aligned} \tag{4.9}$$

Thus, the components of c'c_i are the components of c_i in the program prefixed with c.

For each *this* method call, the arguments and the parameters are equal. For every method call

$$c \triangleright \ x = n_\tau(u)$$

we generate the following constraint. Let t and x be the thread parameter and the first parameter of the method n respectively.

$$c't = \tau \ \wedge \ c'x = u \tag{4.10}$$

If a return statement inside a *this* method call is executed, the argument of the return statement is equal to the returned variable of the *this* method call. For every *this* method call labeled c that calls the method n, we generate the following constraint. Let $\{\overline{c_r}\}$ be the set of return statements of the method n, $Returns_\pi(n)$. Let u_r be the argument of c_r, $arg1_\pi(c_r)$.

$$\bigwedge\nolimits_{c_r \in \{\overline{c_r}\}} (c'c_r \in X) \Rightarrow (c'u_r = x) \tag{4.11}$$

In section 2.2.1, we defined the *execution condition* and the *prior returns* of a label in a specification. The execution condition $cond_\pi(c)$ of a label c is the conjunction of all of the enclosing if or else conditions of c in π. The prior returns $PreReturns_\pi(c)$ of a label c are the set of labels of the return statements before c in π. A *this* method call is executed if and only if its execution condition is satisfied and no prior return statement is executed. For every label c, we generate the following constraint.

$$l \in X \Leftrightarrow \left(cond_\pi(c) \ \wedge \ \bigwedge\nolimits_{c \in PreReturns_\pi(c)} \neg(c' \in X) \right) \tag{4.12}$$

Consider a method call labeled c' inside a *this* method call labeled c. The run-time label of the method is c'c'. The method call c'c' is executed if and only if the method call c is

executed, its execution condition is satisfied and no prior return statement is executed. For every label $c\text{'}c'$, we generate the following constraint.

$$c\text{'}c' \in X \Leftrightarrow \left(c \in X \ \wedge \ c\text{'}cond_{\pi}(c) \ \wedge \ \bigwedge\nolimits_{c'' \in PreReturns_{\pi}(c')} \neg(c\text{'}c'' \in X) \right) \tag{4.13}$$

Note that the execution condition is prefixed with c.

In section 2.2.1, we defined the program order $\rightarrow_{\pi}$. The execution order preserves the program order. For every pair of two *this* method calls labeled c_1 and c_2, if $c_1 \rightarrow_{\pi} c_2$, we generate the following constraint.

$$(c_1 \in X \ \wedge \ c_2 \in X) \Rightarrow c_1 \prec_X c_2 \tag{4.14}$$

For every *this* method call labeled c that calls the method n, for every pair of labels c_1 and c_2 in n, if $c_1 \rightarrow_{\pi} c_2$, we generate the following constraint.

$$(c\text{'}c_1 \in X \ \wedge \ c\text{'}c_2 \in X) \Rightarrow c\text{'}c_1 \prec_X c\text{'}c_2 \tag{4.15}$$

We generate constraints that assert the safety properties of the base objects. For example, for a basic register r, we generate the following definitions according to the semantics of basic

register in Definition 5

$$isXRead_{X,r}(l_R) = l_R \in X \ \wedge \ obj_X(l_R) = r \ \wedge \ name_X(l_R) = read \quad (4.16)$$

$$isXWrite_{X,r}(l_W) = l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = write \quad (4.17)$$

$$NoWriteBetween_{X,r}(l_W, l_R) = \forall l'_W : isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_X l_W \ \vee \ l_R \prec_X l'_W) \quad (4.18)$$

$$isXWriter_{X,r}(l_W, l_R) = isXWrite_{X,r}(l_W) \ \wedge \quad (4.19)$$
$$l_W \prec_X l_R \ \wedge$$
$$NoWriteBetween_{X,r}(l_W, l_R)$$

$$isXRaceFree_{X,r}(l) = \forall l_W : isXWrite_{X,r}(l_W) \Rightarrow l_W \preceq_X l \ \vee \ l \prec_X l_W \quad (4.20)$$

$$isXSequentiallyWritten_r(X) = \forall l \in X : isXWrite_{X,r}(l) \Rightarrow isRaceFree_{X,r}(l) \quad (4.21)$$

and we generate the following constraint

$$isXSequentiallyWritten_r(X) \Rightarrow \quad (4.22)$$
$$\forall l_R : isXRead_{X,r}(l_R) \ \wedge \ isXRaceFree_{X,r}(l_R) \Rightarrow$$
$$\exists l_W : isXWriter_{X,r}(l_W, l_R) \ \wedge$$
$$retv_X(l_R) = arg1_X(l_W)$$

Note that as the set of labels is finite, $\forall$ and $\exists$ can be replaced with $\wedge$ and $\vee$ over all labels.

For each linearizable object o, there should be a linearization L that is equivalent to the executed method calls on o in X, is real-time-preserving and is a member of the sequential specification of o. The linearization order $\prec_L$ is a total order. For every pair of labels l_1 and l_2, we generate the following constraints that assert that $\prec_L$ is irreflexive, transitive and

total.

$$l_1 \prec_L l_2 \Rightarrow \neg(l_2 \prec_L l_1) \tag{4.23}$$

$$(l_1 \prec_L l_2) \ \wedge \ (l_2 \prec_L l_3) \Rightarrow (l_1 \prec_L l_3) \tag{4.24}$$

$$(l_1 \prec_L l_2) \ \vee \ (l_2 \prec_L l_1) \tag{4.25}$$

The linearization L is equivalent to the executed method calls on o in X. A method call is in L if and only if it is in X and is called on o. For every label l, we generate the following constraint.

$$l \in L \Leftrightarrow (l \in X \ \wedge \ obj_X(l) = o) \tag{4.26}$$

$$obj_L(l) = obj_X(l) \ \wedge \ thread_L(l) = thread_X(l) \ \wedge \ name_L(l) = name_X(l) \ \wedge \tag{4.27}$$

$$arg1_L(l) = arg1_X(l) \ \wedge \ arg2_L(l) = arg2_X(l) \ \wedge \ retv_L(l) = retv_X(l) \tag{4.28}$$

The linearization order $\prec_L$ is real-time-preserving. For every pair of labels l_1 and l_2, we generate the following constraint.

$$l_1 \prec_X l_2 \Rightarrow l_1 \prec_L l_2 \tag{4.29}$$

The linearization L is a member of the sequential specification of o. For an atomic register r, we generate the following constraint according to the sequential specification of register (Definition 4).

$$\begin{aligned} &\forall l_R \colon isXRead_{L,r}(l_R) \Rightarrow \\ &\exists l_W \colon isXWriter_{L,r}(l_W, l_R) \ \wedge \\ &\qquad retv_L(l_R) = arg1_L(l_W) \end{aligned} \tag{4.30}$$

Similarly, we can generate constraints for atomic cas register, lock, try-lock and counter

types according to their sequential specifications (Definitions 6, 7, 9, 11).

Finally, we map the test assertion in the input specification to the *negation* of that assertion. As a result, we can use a constraint solver to search for a history that violates the test assertion. If there exists a solution for the constraints, we can construct a bug history as follows. The executed labels is the set of labels l such that $l \in X$. The bug history is the sequence of events of executed labels ordered by the relation $\lhd_X$.

In practice, we apply several optimizations to the constraints presented above. For example, we do not consider invocation and response events for linearizable objects. Thus, we define the execution order on the set of events of the basic objects and the labels of the linearizable objects. Furthermore, we define the linearization history as a sub-history of the execution history. Thus, we restate the constraints that involve the linearization order in terms of the execution order. We can simplify the assertions for the safety of the basic register follows:

$$\begin{aligned} & isXSequentiallyWritten_r(X) \Rightarrow \\ & \forall l_R, l_W : isXRead_{X,r}(l_R) \ \wedge \ isXRaceFree_{X,r}(l_R) \ \wedge \ isXWriter_{X,r}(l_w, l_r) \Rightarrow \\ & \qquad retv_X(l_R) = arg1_X(l_W) \end{aligned} \tag{4.31}$$

Similarly, we can simplify the assertions for the safety of the atomic register follows:

$$\begin{aligned} & \forall l_R, l_W : isXRead_{L,r}(l_R) \ \wedge \ isXWriter_{L,r}(l_W, l_R) \Rightarrow \\ & \qquad retv_L(l_R) = arg1_L(l_W) \end{aligned} \tag{4.32}$$

Our tool. Samand analyzes the input specification and builds a skeleton execution graph. The execution graph is an inlined representation of the concurrent program. Each method call on a linearizable object or return statement is represented as a node in the graph. Each method call on a basic object or on the *this* object is represented as two invocation

and response nodes in the graph. There is a node for every *if* and also every *else* statement. There is an edge to the *if* and *else* nodes from the nodes that they are data-dependent on. There is an edge from *if* and *else* nodes to any statement in their scope. The tool generates the above constraints in SMT2 format and then uses the Z3 SMT solver [17] to solve the constraints. If the constraints are unsatisfiable, then the specification is correct. If the constraints are satisfiable, then the specification is incorrect and the constraint solver will find a model for transaction history that violates the test assertion. The model represents the set of executed events and method calls and their execution order. This information is extracted from the model and added to the execution graph. The resulting graph is topologically sorted and the resulting trace is shown to the user in a graphical user interface. Our tool and some examples are available at [49].

4.4 Experiments

We will now report on running our tool on the two example algorithm specifications. Our first example concerns DSTM.

The context. We believe that DSTM matches the *paper* on DSTM [39]. While we prove that DSTM doesn't satisfy opacity, we have learned from personal communication with Victor Luchangco, one of the DSTM authors, that the *implementation* of DSTM implements more than what was said in the paper and most likely satisfies opacity.

The bug. DSTM provides snapshot isolation by validating the read set (at $R07$) before the read method returns but fails to prevent write skew anomaly. When we run our tool on $(DSTM, P_{WS}, \neg WS)$, we get an execution trace that matches H_{WS}. Figure 4.1(a) presents an illustration of the set of DSTM executions that exhibit the bug. Note that this set is a subset of the set of executions that the bug pattern describes. In Figure 4.1(a), each transaction executes from top to bottom and the horizontal lines denote "barriers", that is, the operations above the line are finished before the operations below the line are started and

otherwise the operations may arbitrarily interleave. For example, $read_{T_1}(2){:}v_0$ should finish execution before $write_{T_2}(2, -v_0)$ but $read_{T_1}(1){:}v_0$ and $read_{T_2}(1){:}v_0$ can arbitrarily interleave. In Figure 4.1(a), T_1 writes to location 1 after T_2 reads from it so T_2 does not abort T_1. T_1 invokes commit and finishes the validation phase ($C01 - C02$) before T_2 effectively commits (executes the *cas* method call at $C03$). The situation is symmetric for transaction T_2. During the validation, the two transactions still see v_0 as the stable value of the two locations; thus, both of them can pass the validation phase. Finally, both of them succeed at *cas*. Note that the counterexample happens when the two commit method calls interleave between $C02$ and $C03$.

The fix. We learned from Victor Luchangco that the *implementation* of DSTM aborts the *writer* transactions of the locations in the read set $rset[T]$ during validation of the commit method call. We model this fix by adding the following lines before $C01$ in DSTM:

foreach $(i \in dom(rset[t]))$ {
  $st := start[i].read()$;
  $t' := st.writer.read()$;
  if $(t \neq t')$
    $state[t'].cas(\mathbb{R}, \mathbb{A})$
}

Those lines prevent H_{WS} because each transaction will abort the other transaction and thus both of them abort.

Our second example concerns McRT.

The context. McRT [69] predates the definition of opacity [28] and wasn't intended to satisfy such a property, as far as we know. Rather, McRT is serializable by design. Still, we prove that McRT doesn't satisfy opacity.

The bug. When we run our tool on $(McRT, P_{WE}, \neg WE)$, we get an execution trace that matches H_{WE} in about 20 minutes. Figure 4.1(b) presents an illustration of the set of executions that exhibit the bug. Like above, this set is a subset of the set of executions that the bug

T_1	T_2
$init_{T_1}()$ $read_{T_1}(1)$:v_0 $read_{T_1}(2)$:v_0	$init_{T_2}()$ $read_{T_2}(1)$:v_0 $read_{T_2}(2)$:v_0
$write_{T_1}(1, -v_0)$ $commit_{T_1}.C01$–$C02$	$write_{T_2}(2, -v_0)$ $commit_{T_2}.C01$–$C02$
$commit_{T_1}.C03$–$C05$	$commit_{T_2}.C03$–$C05$

(a) DSTM counterexamples

T_1	T_2
$init_{T_1}()$ $read_{T_1}(2).R01$–$R03$	$init_{T_2}()$
	$write_{T_2}(2, v_1)$
$read_{T_1}(2).R04$–$R08$	$read_{T_2}(1).R01$–$R03$
$write_{T_1}(1, v_1)$	
	$read_{T_2}(1).R04$–$R08$
$commit_{T_1}.C01$–$C03$	$commit_{T_2}.C01$–$C03$
$commit_{T_1}.C04$	$commit_{T_2}.C04$

(b) McRT counterexamples

Figure 4.1: Counterexamples

pattern describes. Figure 4.1(b) uses the same conventions as Figure 4.1(a). The execution interleaves $write_{T_2}(2, v_1)$ between statements $read_{T_1}(2).R01 - R03$ and $read_{T_1}(2).R04 - R08$ such that the old value of $l[2]$ (unlocked) and the new value of $r[2]$ (the value v_1) are read. Also, $commit_{T_2}.C01 - C03$ are executed before $commit_{T_1}.C04$ such that T_2 finds $l[1]$ locked and aborts. The situation is symmetric for transaction T_1.

The fix. The validation in the commit method ensures that only transactions that have read consistent values can commit; this is the key to why McRT is serializable. Our fix to McRT is to let the read method do validation, that is, to insert a copy of lines $C02 - C04$ between line $R07$ and line $R08$ in McRT.

Let us use Fixed McRT to denote McRT with the above fix. When we run our tool on $(FixedMcRT, P_{WE}, \neg WE)$, our tool determines that the algorithm satisfies the assertion, that is, Fixed McRT doesn't have the write-exposure anomaly. The run takes about 10 minutes.

Note though that in the fixed algorithm, a sequence of writer transactions can make a reader transaction abort an arbitrary number of times. This observation motivated our study of progress for direct-update TM algorithms such as McRT.

Chapter 5

Synchronization Object Program Logic

5.1 Introduction

As we review in section 8.2, the previous works on concurrent program logics support several forms of local reasoning. The reasoning about a thread is done locally on the thread itself and is separate from the reasoning on other threads. For example, the rely/guarantee technique supports local reasoning on a thread when the interference from other threads is known. These logics do not support assertions for the execution order or the linearization order of two method calls in two different threads. These assertions are particularly essential for reasoning about TM algorithms. As we presented in section 3, a concurrent execution of a set of transactions is correct if there is an indistinguishable sequential order of the transactions. The sequential order is determined by the execution order or the linearization order of certain method calls in the transactions. We present a program logic called synchronization object logic (SOL) for reasoning about the behavior of an algorithm based on its syntactic specification. The assertion language of SOL supports execution overlap, execution order and linearization orders of method calls.

SOL provides inference rules that can be conceptually divided into four groups: (1) the standard first-order logic rules, (2) the structure rules that axiomatize the relation of the program structure and the execution, (3) the basic rules that axiomatize the properties of the execution and linearization orders and their interdependence and (4) the synchronization object rules that axiomatize the properties of common synchronization object types.

We define the semantics of the assertion language i.e. we define whether a history models an assertion. Based on the semantics of specifications and the semantics of assertions, we prove the soundness of the logic. SOL derives valid conclusions from valid premises.

In the following chapters, we prove the correctness of the well-known TM algorithm TL2 [18] using SOL. SOL is applicable beyond TM, particularly to algorithms for mutual exclusion. As evidence, we prove the mutual exclusion property of the Dekker algorithm in this chapter.

In the first section of this chapter, we showcase the logic with a simple example. We present a simple specification, a simple lemma about the specification and simplified versions of the inference rules. We illustrate the proof of the lemma using the inference rules. In the next subsections, we consider the full logic. We define the assertion language and the semantics of assertions. Then, we define the four groups of inference rules. Then, we formalize and prove the soundness of the logic. Finally, we present the proof the correctness of the Dekker algorithm using the inference rules.

5.2 Simple Example

We introduce the program logic SOL via a simple example. In this section, we present, first, an example specification in a subset of the specification language, then, the simplified program logic and finally, the deduction of a lemma for the example specification.

$\mathcal{T}$:				
	$lock$: $Lock$			
	$clock$: $SCounter$			
	ver: $BasicRegister$			
$\mathcal{P}$:				
$L_1\triangleright$	$lock.lock_{T_1}()$		$L_2\triangleright$	$lock.lock_{T_2}()$
$C_1\triangleright$	$v_1 = clock.iaf_{T_1}()$	$\|\|$	$C_2\triangleright$	$v_2 = clock.iaf_{T_2}()$
$R_1\triangleright$	$ver.write_{T_1}(v_1)$		$R_2\triangleright$	$ver.write_{T_2}(v_2)$
$U_1\triangleright$	$lock.unlock_{T_1}()$		$U_2\triangleright$	$lock.unlock_{T_2}()$

Figure 5.1: Example Specification π

5.2.1 Algorithm Specification

Figure 5.1 specifies a simple algorithm that updates a register to ascending version numbers. In fact, it is a miniature version of the TL2 commit procedure. This specification has two sections: the type declaration section at the top and the concurrent program section at the bottom. In general, a specification can have a procedure definition section and call procedures that we postpone to the next section.

The type declaration section declares the *type* of each synchronization object used by the concurrent program. Three object types are used in this program: lock *Lock*, strong counter *SCounter* and basic register *BasicRegister*. Lock and strong counter are linearizable object types and basic register is a basic object type. In the general sense, linearizable objects can maintain *consistency* even if they are accessed *concurrently* while basic objects maintain consistency if they are not accessed concurrently. A register has two methods: *write* and *read*. For example, $r.write(v)$ writes the value v to r, while $x = r.read()$ reads the value of r and binds x to that value. The language enforces unique binding for variables. A lock has two methods *lock* and *unlock* that lock and unlock it respectively. A strong counter has two methods: *read* and *iaf* (increment-and-fetch). For a strong counter c, $x = c.read()$ reads the value of c and binds x to that value and $x = c.iaf()$ increments and then reads the value of c and binds x to that value. The objects *lock*, *clock* and *ver* are declared of *Lock*, *SCounter*, and *BasicRegister* types.

The second section is the concurrent program. It is the parallel composition of a set of sequential programs. In this specification, there are two sequential programs where every statement is a method call. A method call is of the form $l \triangleright\ x = o.n_\tau(u)$ where l is the unique label of the method call. We define the following functions on labels that are immediately derived from the specification. obj_π maps l to the receiving object o, $name_\pi$ maps l to the method name n, $thread_\pi$ maps l to the calling thread identifier τ, $arg1_\pi$ maps l to the first argument u (that is either a variable x or a value v), and $retv_\pi$ maps l to the return variable x. The function $cond_\pi$ maps l to the *enclosing condition* of the method call labeled l. In this specification, we do not have if-then-else statements, therefore, $cond_\pi(l) = true$ for every label l. Every specification π, defines a *program order* $\rightarrow_\pi$ on the labels. Intuitively, $l_1 \rightarrow_\pi l_2$ means that the specification requires that if both l_1 and l_2 are executed, then l_1 must be executed before l_2. In this specification, we assume sequential consistency. Therefore, the program order $\rightarrow_\pi$ simply represents the order of labels in the program. We postpone relaxed order of method calls to next later section.

5.2.2 Program Logic

Consider the two method calls labeled R_1 and R_2 in the specification (Figure 5.1). We will prove the following theorem that states that if the version that R_1 writes is less than the version that R_2 writes, then R_1 is executed before R_2. Although the statement of the lemma is simple, similar to the TM correctness assertions, it involves execution order and its proof involves linearization order of synchronization objects.

Lemma 35. $\pi, \cdot \vdash (arg1(R_1) < arg1(R_2)) \Rightarrow (R_1 \prec R_2)$.

Let us have an informal proof of the lemma first. We use the following five rules. First, the *program-order-preservation* property states that the program order is preserved in the execution order. Second, the *real-time-preservation* property states that the execution order is preserved in the linearization order. Third, the *execution-linearization-transitivity* property states that if l_1 is executed before l_2, l_2 is linearized before l_3 and l_3 is executed before

$$\textsc{Control}\ \frac{}{\pi, \Gamma \vdash exec(l) \Leftrightarrow cond_\pi(l)}$$

$$\textsc{Id}\ \frac{\begin{array}{c} obj_\pi(l) = o \qquad name_\pi(l) = n \\ thread_\pi(l) = \tau \qquad arg1_\pi(l) = u \qquad retv_\pi(l) = x \\ \pi, \Gamma \vdash exec(l) \end{array}}{\begin{array}{c} \pi, \Gamma \vdash obj(l) = o \ \wedge \ name(l) = n \ \wedge \\ thread(l) = \tau \ \wedge \ arg1(l) = u \ \wedge \ retv(l) = x \end{array}}$$

$$\text{P2X}\ \frac{l_1 \rightarrow_\pi l_2 \qquad \pi, \Gamma \vdash exec(l_1) \qquad \pi, \Gamma \vdash exec(l_2)}{\pi, \Gamma \vdash l_1 \prec l_2}$$

$$\textsc{Src}\ \frac{\pi, \Gamma \vdash exec(l) \qquad \pi, \Gamma \vdash obj(l) = o \qquad \pi, \Gamma \vdash name(l) = n \qquad Calls_\pi(o, n) = \{\overline{l_i}\}}{\pi, \Gamma \vdash \bigvee_{i=1..n} l = l_i}$$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 5.2: Structure Inference Rules.

$$\text{X2L}\ \frac{\mathcal{T}(o) \in LT \qquad \pi, \Gamma \vdash obj(l) = obj(l') = o \qquad \pi, \Gamma \vdash l \prec l'}{\pi, \Gamma \vdash l \prec_o l'}$$

$$\textsc{XLTrans}\ \frac{\pi, \Gamma \vdash l_1 \prec l_2 \qquad \pi, \Gamma \vdash l_2 \prec_o l_3 \qquad \pi, \Gamma \vdash l_3 \prec l_4}{\pi, \Gamma \vdash l_1 \prec l_4}$$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 5.3: Basic inference rules.

CountSeq

$$\frac{\begin{array}{c}\mathcal{T}(o) = SCounter \\ \pi, \Gamma \vdash exec(l_1) \ \wedge \ obj(l_1) = o \ \wedge \ name(l_1) = iaf \\ \pi, \Gamma \vdash exec(l_2) \ \wedge \ obj(l_2) = o \\ \pi, \Gamma \vdash retv(l_1) < retv(l_2)\end{array}}{\pi, \Gamma \vdash l_1 \prec_o l_2}$$

LockUnlockPair

$$\frac{\begin{array}{c}\mathcal{T}(o) = Lock \\ \pi, \Gamma \vdash isOwnerRespect(o) \\ \pi, \Gamma \vdash isLock_o(l_{l_1}) \qquad \pi, \Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_{lo}(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \wedge \\ isLock_{lo}(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

$isLock_o(l) \Leftrightarrow$
$\quad exec(l) \ \wedge \ obj(l) = o \ \wedge \ name(l) = lock$
$isUnlock_o(l) \Leftrightarrow$
$\quad exec(l) \ \wedge \ obj(l) = o \ \wedge \ name(l) = unlock$
$isOwnerRespect(o) \Leftrightarrow$
$\quad \forall \ell: isUnlock_o(\ell) \Rightarrow \exists \ell':$
$\quad\quad isLock_o(\ell') \wedge$
$\quad\quad thread(\ell') = thread(\ell) \ \wedge \ \ell' \prec \ell \wedge$
$\quad\quad \forall \ell'':$
$\quad\quad\quad (isUnLock_o(\ell'') \wedge$
$\quad\quad\quad thread(\ell'') = thread(\ell))$
$\quad\quad \Rightarrow$
$\quad\quad\quad \ell'' \prec \ell' \ \vee \ \ell \preceq \ell''$

Each rule has the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 5.4: Synchronization Object Inference Rules.

l_4, then l_1 is executed before l_4. Forth, the *lock-unlock-pair* property states that if ownership of a lock l is respected and a *lock* method call on l (by a thread T_1) is linearized before an *unlock* method call on l (by a thread T_2), then an *unlock* method call on l by T_1 is linearized before a *lock* method call on l by T_2. Intuitively, ownership for a lock l is respected, if and only if every thread unlocks l only if it has already locked l and has not unlocked l since it has locked l. This specification π trivially respects ownership for its *lock* object. Fifth, the *count-sequence* property states that for a strong counter o, if the return value of an *iaf* method call on o is less than the return value of another method call on o, then the former is linearized before the latter.

We assume that (1) The argument of R_1 is less than the argument of R_2 and show that R_1 is executed before R_2. From the specification π, we have that (2) The argument of R_1 is the return value of C_1 and (3) the argument of R_2 is the return value of C_2. Thus, from [1], [2] and [3], we have that (4) the return value of C_1 is less than the return value of C_2. From π, we have that (5) C_1 and C_2 are *iaf* method calls on *clock* that is a strong counter. Thus, by count-sequence property on [5] and [4], we have that (6) C_1 is linearized before C_2. From π, we have (7) L_1 is before C_1 in the program and (8) C_1 is before U_2 in the program. By program-order-preservation on [7] and [8], we have that (9) L_1 is executed before C_1 and (10) C_2 is executed before U_2. By execution-linearization-transitivity property on [9], [6] and [10], we can conclude that (11) L_1 is executed before U_2. From π, we have (12) L_1 and U_2 are respectively *lock* and *unlock* method calls by threads T_1 and T_2 on the object *lock* that is of the linearizable type *Lock*. By the real-time-preservation property on [11], we have that (13) L_1 is linearized before U_2. By the lock-unlock-pair property on [12] and [13], we have that (14) an *unlock* method call by T_1 is linearized before a *lock* method call by T_2. From π, we have that (15) The *unlock* method call by T_1 is U_1 and (16) The *lock* method call by T_2 is L_2. Thus, from [14], [15] and [16], we have that (17) U_1 is linearized before L_2. From π, we have (18) R_1 is before U_1 in the program and (19) L_1 is before R_2 in the program. From the program-order-preservation property on [18] and [19], we have that (20) R_1 is executed

before U_1 and (21) L_2 is executed before R_2. By the transitivity property on [20], [17] and [21], we have that R_1 is executed before R_2.

Now, let us introduce our synchronization object logic (SOL) and formalize the proof. The judgments of SOL are of the form $\pi, \Gamma \vdash \mathcal{A}$, where π is a specification, Γ is a list of assertions and $\mathcal{A}$ is an assertion. We use $\cdot$ to denote the empty list of assertions. Intuitively, a judgment $\pi, \Gamma \vdash \mathcal{A}$ states that in the context of the assertions Γ, the specification π has the property $\mathcal{A}$. The assertions are first-order logic assertions that involve the unary predicate *exec*, the binary predicates $\prec$ (*execution order*) and $\prec_o$ (*linearization order* of linearizable object o) and functions *obj*, *name*, *thread*, $arg1$ and *retv*. The assertion $exec(l)$ states that the method call labeled l is executed. The assertion $l_1 \prec l_2$ states that l_1 is executed before l_2. Any concurrent execution on a linearizable object is equivalent to a correct sequential execution. The total order of method calls in the equivalent sequential execution is called the linearization order. For every linearizable object o, the assertion $l_1 \prec_o l_2$ states that l_1 is before l_2 in the linearization order of o. As π declares *lock* and *clock* as instances of linearizable types, the linearization orders of *lock* and *clock* are denoted by $\prec_{lock}$ and $\prec_{clock}$. We also use the equivalence relation on expressions and labels. The functions $obj(l)$, $name(l)$, $thread(l)$, $arg1(l)$, and $retv(l)$ map a label l to the receiving object, method name, calling thread identifier, the first argument and the return value of the method call labeled l.

Lemma 35 expresses a property of every execution of π, yet the soundness of SOL makes us able to prove it by reasoning about π alone. We consider an arbitrary execution of the specification. Given some facts about an execution, the inference rules let us derive more facts about that execution. SOL has four sets of inference rules: classical first-order logic inference rules, structure inference rules that axiomatize the association of the specification and the assertions, basic inference rules that axiomatize the properties of the execution and linearization orders and their interdependence and synchronization object inference rules that axiomatize the properties of common synchronization object types. We showcase a subset

of structure inference rules in Figure 5.2, a subset of basic inference rules in Figure 5.3, and a subset of synchronization object inference rules in Figure 5.4.

The rule CONTROL states that a method call is executed if and only if its enclosing condition is satisfied. The introduction rule ID states that the components (object, name, etc.) of a method call in the execution originate from the components of the method call in the program. The rule P2X states the program-order-preservation property. If a method call l_1 is ordered before a method call l_2 in the program, and methods l_1 and l_2 are executed, then l_1 is executed before l_2. The rule SRC intuitively states that every executed method originates from a call site in the specification. Let $Calls_\pi(o, n)$ denote the set of labels of call sites where method name n is called on the object name o in the specification π. If the object and the name of an executed method call labeled l are o and n respectively, then l is equal to one of the labels in $Calls_\pi(o, n)$. For presentation purposes, this small example does not involve procedure calls and hence the rules CONTROL, ID, and SRC are simplified.

The rule X2L states the real-time-preservation property. The execution order of two method calls on a linearizable object is preserved in the linearization order. LT denotes the set of linearizable object types. The rule XLTRANS states the execution-linearization-transitivity property defined above. Similarly, the rule LOCKUNLOCKPAIR and the rule COUNTSEQ state the the lock-unlock-pair and count-sequence properties defined above. The rule LOCKUNLOCKPAIR is derived from the fact that if the ownership of a lock is respected, its linearization order is a sequence of pairs of *lock* and *unlock* method calls by the same thread. The rule COUNTSEQ is derived from the fact that the return value of method calls in the linearization order of a strong counter is non-decreasing.

5.2.3 Deduction

Now, let us see how the above informal reasoning can be formalized using SOL inference rules. Let

$$\Gamma = arg1(R_1) < arg1(R_2) \tag{5.1}$$

Based on the classical condition introduction rule, to prove Lemma 35, we need to show that

$$\pi, \Gamma \vdash R_1 \prec R_2 \tag{5.2}$$

From 5.1, we have

$$\pi, \Gamma \vdash arg1(R_1) < arg1(R_2) \tag{5.3}$$

As mentioned before, there is no if-then-else in this specification; therefore, the enclosing condition of every label is trivially *true*. Thus, by the rule CONTROL, we have

$$\pi, \Gamma \vdash exec(L_1) \tag{5.4}$$

$$\pi, \Gamma \vdash exec(C_1) \tag{5.5}$$

$$\pi, \Gamma \vdash exec(R_1) \tag{5.6}$$

$$\pi, \Gamma \vdash exec(U_1) \tag{5.7}$$

$$\pi, \Gamma \vdash exec(L_2) \tag{5.8}$$

$$\pi, \Gamma \vdash exec(C_2) \tag{5.9}$$

$$\pi, \Gamma \vdash exec(R_2) \tag{5.10}$$

$$\pi, \Gamma \vdash exec(U_2) \tag{5.11}$$

From the rule ID on 5.6, 5.10, 5.5, 5.9, and the specification π, we have

$$\pi, \Gamma \vdash arg1(R_1) = v_1 \tag{5.12}$$

$$\pi, \Gamma \vdash arg1(R_2) = v_2 \tag{5.13}$$

$$\pi, \Gamma \vdash retv(C_1) = v_1 \tag{5.14}$$

$$\pi, \Gamma \vdash retv(C_2) = v_2 \tag{5.15}$$

From the symmetry and transitivity of equivalence on [5.12], [5.13], [5.14], [5.15], we have

$$\pi, \Gamma \vdash arg1(R_1) = retv(C_1) \tag{5.16}$$

$$\pi, \Gamma \vdash arg1(R_2) = retv(C_2) \tag{5.17}$$

By substitution of 5.16 and 5.17 on [5.3], we have

$$\pi, \Gamma \vdash retv(C_1) < retv(C_2) \tag{5.18}$$

By the rule ID on 5.5, and the specification π, we have

$$\pi, \Gamma \vdash obj(C_1) = clock \tag{5.19}$$

$$\pi, \Gamma \vdash name(C_1) = iaf \tag{5.20}$$

By the rule ID on 5.9, and the specification π, we have

$$\pi, \Gamma \vdash obj(C_2) = clock \tag{5.21}$$

From rule COUNTSEQ on 5.5, 5.19, 5.20, 5.9, 5.21, 5.18, we have

$$\pi, \Gamma \vdash C_1 \prec_{clock} C_2 \tag{5.22}$$

that is C_1 is linearized before C_2. The next step is to use rule P2X. From π, we have

$$L_1 \rightarrow_\pi C_1 \tag{5.23}$$

$$C_2 \rightarrow_\pi U_2 \tag{5.24}$$

By the rule P2X on 5.23, 5.4 and 5.5, we have

$$\pi, \Gamma \vdash L_1 \prec C_1 \tag{5.25}$$

Similarly, by the rule P2X on 5.24, 5.9 and 5.11, we have

$$\pi, \Gamma \vdash C_2 \prec U_2 \tag{5.26}$$

By the rule XLTrans on 5.25, 5.22 and 5.26, we have

$$\pi, \Gamma \vdash L_1 \prec U_2 \tag{5.27}$$

By the rule Id on 5.4, and the specification π, we have

$$\pi, \Gamma \vdash obj(L_1) = lock \tag{5.28}$$

$$\pi, \Gamma \vdash name(L_1) = lock \tag{5.29}$$

$$\pi, \Gamma \vdash thread(L_1) = T_1 \tag{5.30}$$

Similarly, by the rule Id on 5.11, and the specification π, we have

$$\pi, \Gamma \vdash obj(U_2) = lock \tag{5.31}$$

$$\pi, \Gamma \vdash name(U_2) = unlock \tag{5.32}$$

$$\pi, \Gamma \vdash thread(U_2) = T_2 \tag{5.33}$$

From rule X2L on 5.27, 5.28 and 5.31, we have

$$\pi, \Gamma \vdash L_1 \prec_{lock} U_2 \tag{5.34}$$

Now, we use the rule LockUnlockPair. The proof of ownership respect can be done using the presented rules. For the sake of brevity, we skip the proof of ownership respect.

$$\pi, \Gamma \vdash isOwnerRespecting(lock) \tag{5.35}$$

From the definition of *isLock* on 5.4, 5.28 and 5.29, we have

$$\pi, \Gamma \vdash isLock_{lock}(L_1) \tag{5.36}$$

From the definition of *isUnlock* on 5.11, 5.31 and 5.32, we have

$$\pi, \Gamma \vdash isUnlock_{lock}(U_2) \tag{5.37}$$

By the rule LockUnlockPair on 5.35, 5.36, 5.37, 5.34, and then substitution with 5.30 and 5.33, we have

$$\begin{aligned} \pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2} : & \; isUnlock_{lock}(\ell_{u_1}) \; \wedge \; thread(\ell_{u_1}) = T_1 \; \wedge \\ & \; isLock_{lock}(\ell_{l_2}) \; \wedge \; thread(\ell_{l_2}) = T_2 \; \wedge \\ & \; \ell_{u_1} \prec_{lock} \ell_{l_2} \end{aligned} \tag{5.38}$$

After skolemization of ℓ_{u_1} and ℓ_{l_2} with l_{u_1} and l_{l_2}, we have

$$\pi, \Gamma \vdash isUnlock_{lock}(l_{u_1}) \tag{5.39}$$

$$\pi, \Gamma \vdash thread(l_{u_1}) = T_1 \tag{5.40}$$

$$\pi, \Gamma \vdash isLock_{lock}(l_{l_2}) \tag{5.41}$$

$$\pi, \Gamma \vdash thread(l_{l_2}) = T_2 \tag{5.42}$$

$$\pi, \Gamma \vdash l_{u_1} \prec_{lock} l_{l_2} \tag{5.43}$$

From the definition of $isUnlock$ on 5.39, we have

$$\pi, \Gamma \vdash exec(l_{u_1}) \tag{5.44}$$

$$\pi, \Gamma \vdash obj(l_{u_1}) = lock \tag{5.45}$$

$$\pi, \Gamma \vdash name(l_{u_1}) = unlock \tag{5.46}$$

From π, we have

$$Calls_{\pi}(lock, unlock) = \{U_1, U_2\} \tag{5.47}$$

By the rule Src on 5.44, 5.45, 5.46, and 5.47, we have

$$\pi, \Gamma \vdash l_{u_1} = U_1 \ \vee \ l_{u_1} = U_2 \tag{5.48}$$

Using negation introduction, from 5.33 and 5.40, we have

$$\pi, \Gamma \vdash \neg(l_{u_1} = U_2) \tag{5.49}$$

By disjunction syllogism on 5.48 and 5.49, we have

$$\pi, \Gamma \vdash l_{u_1} = U_1 \tag{5.50}$$

Similarly, using the rule Src, we can show that

$$\pi, \Gamma \vdash l_{l_2} = L_2 \tag{5.51}$$

By substitution of 5.50 and 5.51 to 5.43, we have

$$\pi, \Gamma \vdash U_1 \prec_{lock} L_2 \tag{5.52}$$

From π, we have

$$R_1 \rightarrow_\pi U_1 \tag{5.53}$$

$$L_2 \rightarrow_\pi R_2 \tag{5.54}$$

By the rule P2X on 5.53, 5.6 and 5.7, we have

$$\pi, \Gamma \vdash R_1 \prec U_1 \tag{5.55}$$

By the rule P2X on 5.54, 5.8 and 5.10, we have

$$\pi, \Gamma \vdash L_2 \prec R_2 \tag{5.56}$$

By the rule XLTrans on 5.55, 5.52, and 5.56, we have

$$\pi, \Gamma \vdash R_1 \prec R_2 \tag{5.57}$$

5.3 Assertion Language

Now, we define the assertion language of the logic. We first define label variables as follows:

$$\begin{array}{rcll} \ell \in \mathit{LabelVar} & ::= & \{\ell_1, \ell_2, ...\} & \text{Variable Label} \\ \not{l} \in \mathit{Label} & := & l \mid \ell & \text{Label} \end{array}$$

Consider a specification $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$ where $\mathbb{O} = \{o \mid \mathcal{T}_{base}(o) \in LT\}$.

We define the set of assertions $\mathcal{A}$ for the specification π as follows:

$$
\begin{array}{rcll}
e & ::= & obj(\ell) \mid name(\ell) \mid thread(\ell) \mid & \text{Element} \\
 & & arg1(\ell) \mid arg2(\ell) \mid retv(\ell) \mid & \\
 & & initOf(\tau) \mid commitOf(\tau) \mid & \\
 & & o \mid n \mid \varsigma\text{'}x \mid v \mid \varsigma\text{'}t \mid T & \\
\mathcal{R} & ::= & e = e \mid e < e \mid & \text{Atomic Assertion} \\
 & & \ell = \ell' \mid c = c \mid & \\
 & & exec(\ell) \mid & \\
 & & \ell \prec \ell' \mid \ell \sim \ell' \mid \ell \prec_o \ell' \text{ where } o \in \mathbb{O} \mid & \\
 & & \tau \nprec \tau' \mid & \\
\mathcal{A} & ::= & \neg\mathcal{A} \mid \mathcal{A} \wedge \mathcal{A} \mid & \text{Assertion} \\
 & & \forall \ell \colon \mathcal{A} \mid & \\
 & & \forall t \colon \mathcal{A} \mid & \\
 & & \mathcal{R} &
\end{array}
$$

We consider the set of closed formulas.

The set of functions F and predicates Pr are as follows.

$$f \in F = \{obj, name, thread, arg1, arg2, retv, initOf, commitOf\}$$

$$pr \in Pr = \{=, <, exec, \prec, \sim, \prec_o, \nprec\} \text{ where } o \in \mathbb{O}$$

The functions *obj*, *name*, *thread*, $arg1$, $arg2$ and *retv* map a label to its object, method name, first and second argument and return value. The function *initOf* maps each transaction to its *init* method call. The function *commitOf* maps each committed transaction to its *commit* method call. The assertion $exec(\ell)$ asserts that ℓ is executed. The assertion $\ell \prec \ell'$ asserts that ℓ is executed before ℓ'. The assertion $\ell \prec \ell'$ asserts that ℓ is executed concurrent with ℓ'. The assertion $\ell \sim \ell'$ asserts that ℓ is executed concurrent to ℓ'. The assertion $\ell \prec_o \ell'$

asserts that ℓ is linearized before ℓ' on object o. The assertion $\tau \prec\!\!\!\prec \tau'$ asserts that (all the labels of) thread τ is executed before (all the labels of) thread τ'.

We define the following abbreviations:

$$\begin{aligned}
\mathcal{A} \vee \mathcal{A}' &= \neg((\neg\mathcal{A}) \wedge (\neg\mathcal{A}')) \\
\exists \ell\colon \mathcal{A} &= \neg(\forall \ell\colon (\neg A)) \\
e \leq e' &= e \leq e' \vee e = e' \\
\ell \preceq \ell' &= \ell \prec \ell' \vee \ell = \ell' \\
\tau \preceq\!\!\!\prec \tau' &= \tau \prec\!\!\!\prec \tau' \vee \tau = \tau'
\end{aligned}$$

5.4 Assertion Semantics

Now, we define the semantics of assertions. We define the models relation $\models$ between execution histories $\mathcal{X}$ and assertions $\mathcal{A}$.

Let $\mathcal{X} = (X, \sigma, \mathcal{L})$ and $X' = \sigma(X)$. We define the mapping function $\alpha_{\mathcal{X}}$ as follows:

$\alpha_{\mathcal{X}} = \{$

$obj \mapsto obj_{X'}, name \mapsto name_{X'}, thread \mapsto thread_{X'},$

$arg1 \mapsto arg1_{X'}, arg2 \mapsto arg2_{X'}, retv \mapsto retv_{X'},$

$initOf \mapsto initOf, commitOf \mapsto commitOf,$

$= \mapsto =, \ < \mapsto <,$

$exec \mapsto \lambda x.\ x \in X', \ \prec \mapsto \prec_{X'}, \ \sim \mapsto \sim_{X'}, \ \prec_o \mapsto \prec_{\mathcal{L}(\sigma(o))}, \ \prec\!\!\!\prec \mapsto \prec\!\!\!\prec_{X'},$

$o \mapsto \sigma(o),\ n \mapsto n,\ u \mapsto \sigma(u),\ \tau \mapsto \sigma(\tau),$

$c \mapsto c,\ l \mapsto l$

$\}$

The mapping function $\alpha_{\mathcal{X}}$ maps functions and predicates in the assertion language to concrete functions and predicates in the execution $\mathcal{X}$. For example, the function obj is mapped to the function $obj_{X'}$. The mapping function $\alpha_{\mathcal{X}}$ also maps every variable to its value in the

execution $\mathcal{X}$. Thus, x is mapped to $\sigma(x)$. Note that as an object o can be an element of an array that is indexed by a variable, o is mapped to $\sigma(o)$. The function $\alpha_{\mathcal{X}}$ is lifted to closed atomic assertions $\mathcal{R}$ by applying $\alpha_{\mathcal{X}}$ inductively to the structure of $\mathcal{R}$.

Definition 15. *The model relation $\models$ is defined inductively as follows:*

Base case:

$\mathcal{X} \models \mathcal{R}$ *iff* ($\mathcal{R}$ *is closed and* $\alpha_{\mathcal{X}}(\mathcal{R})$)

Inductive case:

$\mathcal{X} \models (\neg\mathcal{A})$ *iff* $\neg(\mathcal{X} \models \mathcal{A})$,

$\mathcal{X} \models (\mathcal{A}_1 \wedge \mathcal{A}_2)$ *iff* $(\mathcal{X} \models \mathcal{A}_1) \wedge (\mathcal{X} \models \mathcal{A}_2)$,

$\mathcal{X} \models (\forall \ell\colon \mathcal{A})$ *iff* $\bigwedge_{l \in Labels(X')} (\mathcal{X} \models \mathcal{A}[\ell := l])$,

$\mathcal{X} \models (\forall t\colon \mathcal{A})$ *iff* $\bigwedge_{T \in Threads(X')} (\mathcal{X} \models \mathcal{A}[t := T])$.

If $\mathcal{X} \models \mathcal{A}$, we say that $\mathcal{X}$ models $\mathcal{A}$.

5.5 Inference Rules

We now present the inference rules of SOL.

The judgments are of the form $\pi, \Gamma \vdash \mathcal{A}$ read assertion $\mathcal{A}$ is derived from the assumption assertions Γ for the specification π. The context Γ is defined as follows:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma; \mathcal{A} \qquad \text{Context}$$

We present the classical first-order logic rules, the structure inference rules, the basic inference rules, and the synchronization object inference rules.

5.5.1 Classical First-order Logic Inference Rules

The classical inference rules are presented in Figure 5.5. The derived classical inference rules are presented in Figure 5.6.

The equivalence and arithmetic Rules are presented in Figure 5.7. The derived equivalence and arithmetic Rules are presented in Figure 5.8.

$$\textsc{Premise}\ \frac{}{\pi, \Gamma; \mathcal{A}; \Gamma' \vdash \mathcal{A}}$$

$$\textsc{NegIntro}\ \frac{\pi, \Gamma; \mathcal{A} \vdash \mathcal{A}' \quad \pi, \Gamma; \mathcal{A} \vdash \neg\mathcal{A}'}{\pi, \Gamma \vdash \neg\mathcal{A}}$$

$$\textsc{ConjIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A} \qquad \pi, \Gamma \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'}$$

$$\textsc{ExcMid}\ \frac{}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \neg\mathcal{A}}$$

$$\textsc{ConjElimL}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}} \qquad \textsc{ConjElimR}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \wedge\ \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}'}$$

$$\textsc{NegElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \quad \pi, \Gamma \vdash \neg\mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}'}$$

$$\textsc{DisjIntroL}\ \frac{\pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}'} \qquad \textsc{DisjIntroR}\ \frac{\pi, \Gamma \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}'}$$

$$\textsc{UnivIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A}[\ell := l] \quad l \notin \Gamma}{\pi, \Gamma \vdash \forall \ell\colon \mathcal{A}}$$

$$\textsc{DisjElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}' \quad \pi, \Gamma; \mathcal{A} \vdash \mathcal{A}'' \quad \pi, \Gamma; \mathcal{A}' \vdash \mathcal{A}''}{\pi, \Gamma \vdash \mathcal{A}''}$$

$$\textsc{UnivElim}\ \frac{\pi, \Gamma \vdash \forall \ell\colon \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}[\ell := l]}$$

$$\textsc{CondIntro}\ \frac{\pi, \Gamma; \mathcal{A} \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}'}$$

$$\textsc{ExistIntro}\ \frac{\pi, \Gamma \vdash \mathcal{A}[\ell := l]}{\pi, \Gamma \vdash \exists \ell\colon \mathcal{A}}$$

$$\textsc{CondElim}\ \frac{\pi, \Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}' \quad \pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}'}$$

$$\textsc{ExistElim}\ \frac{\pi, \Gamma \vdash \exists \ell\colon \mathcal{A} \quad l \notin \Gamma \quad \pi, \Gamma; \mathcal{A}[\ell := l] \vdash \mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}'}$$

Figure 5.5: Classical Inference Rules

$$\textsc{DisjSyllL}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}' \quad \pi, \Gamma \vdash \neg\mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}'} \qquad \textsc{DisjSyllR}\ \frac{\pi, \Gamma \vdash \mathcal{A} \ \vee\ \mathcal{A}' \quad \pi, \Gamma \vdash \neg\mathcal{A}'}{\pi, \Gamma \vdash \mathcal{A}}$$

$$\textsc{CondElim'}\ \frac{\pi, \Gamma \vdash \mathcal{A} \Rightarrow \mathcal{A}' \quad \pi, \Gamma \vdash \neg\mathcal{A}'}{\pi, \Gamma \vdash \neg\mathcal{A}}$$

Figure 5.6: Derived Classical Inference Rules

$$\textsc{LRefl}\ \frac{}{\pi, \Gamma \vdash l = l} \qquad \textsc{ERefl}\ \frac{}{\pi, \Gamma \vdash e = e} \qquad \textsc{Zero}\ \frac{}{\pi, \Gamma \vdash \neg(1 = 0)}$$

$$\textsc{LSubs}\ \frac{\pi, \Gamma \vdash l = l' \quad \pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}[l := l']} \qquad \textsc{LSubs}\ \frac{\pi, \Gamma \vdash e = e' \quad \pi, \Gamma \vdash \mathcal{A}}{\pi, \Gamma \vdash \mathcal{A}[e := e']}$$

Figure 5.7: Equivalence and Arithmetic Rules

$$\textsc{LSym}\ \frac{\pi, \Gamma \vdash l = l'}{\pi, \Gamma \vdash l' = l} \qquad \textsc{LTrans}\ \frac{\pi, \Gamma \vdash l = l' \quad \pi, \Gamma \vdash l' = l''}{\pi, \Gamma \vdash l = l''}$$

$$\textsc{ESym}\ \frac{\pi, \Gamma \vdash e = e'}{\pi, \Gamma \vdash e' = e} \qquad \textsc{ETrans}\ \frac{\pi, \Gamma \vdash e = e' \quad \pi, \Gamma \vdash e' = e''}{\pi, \Gamma \vdash e = e''}$$

Figure 5.8: Derived Equivalence and Arithmetic Rules

5.5.2 Structure Inference Rules

The structure inference rules that axiomatize the relation of the program structure and the execution. The structure inference rules are presented in Figures 5.9 The derived structure inference rules are presented in Figure 5.10. The derived inference rules can be derived from the basic rules. Please see Section 10.4.2 for notes on the derivation of the derived rules.

The rule ID states that components of method calls in the history originate from components of method calls in the program. The object, arguments and other components of an executed method call labeled ς'c can be derived from prefixing the object, arguments and other components of the method call annotated with c in the program with the pre-label ς. Note that the pre-label ς is a constant c' when the method call c is executed inside the body of a *this* method call annotated with c'. The pre-label ς is ϵ when c is the annotation of a *this* method call.

The rule SRC states that every executed method originates from a call site in the program. If a method n on an object with the base name ϕ is executed, it is from one of the call sites where n is called on ϕ in the program.

The rule OCONTROL states when a *this* method call is executed. A *this* method call is executed if an only if its execution condition is satisfied.

The rule ICONTROL states when a method call in a *this* method call is executed. A method call (annotated with) c' in a *this* method call (annotated with) c is executed if and only if c is executed, the execution condition of c' is satisfied and no return statement before c' is executed.

The rule P2X states that the program order is preserved in the execution order. If a method call annotated with c_1 is ordered before a method call annotated with c_2 in the program, and methods labeled ς'c_1 and ς'c_2 are executed, then ς'c_1 is executed before ς'c_2.

The rule OX2IX states that the execution order of two *this* method calls implies the execution order of method calls in their bodies. If a *this* method call c_1 is executed before

ID

$$\frac{\begin{array}{c} obj_\pi(c) = \theta \qquad name_\pi(c) = n \\ thread_\pi(c) = \tau \qquad arg_\pi(c) = u \qquad retv_\pi(c) = x \\ \pi, \Gamma \vdash exec(\varsigma\text{'}c) \end{array}}{\begin{array}{c} \pi, \Gamma \vdash obj(\varsigma\text{'}c) = \varsigma\text{'}\theta \wedge \\ name(\varsigma\text{'}c) = n \ \wedge \ thread(\varsigma\text{'}c) = \varsigma\text{'}\tau \wedge \\ arg(\varsigma\text{'}c) = \varsigma\text{'}u \ \wedge \ retv(\varsigma\text{'}c) = \varsigma\text{'}x \end{array}}$$

SRC

$$\frac{\begin{array}{c} \pi, \Gamma \vdash exec(\varsigma\text{'}c) \\ \pi, \Gamma \vdash obj(\varsigma\text{'}c) = \theta \qquad \pi, \Gamma \vdash name(\varsigma\text{'}c) = n \\ Calls_\pi(basename(\theta), n) = \{\overline{c_i}\} \end{array}}{\pi, \Gamma \vdash \bigvee_{i=1..n} c = c_i}$$

OCONTROL

$$\frac{c \in Labels(\mathcal{P})}{\pi, \Gamma \vdash exec(c) \Leftrightarrow cond_\pi(c)}$$

ICONTROL

$$\frac{\begin{array}{c} Labels(name_\pi(c)) = \{\overline{c_i}\} \\ PreReturns_\pi(c') = \{\overline{c_r}\} \end{array}}{\begin{array}{c} \pi, \Gamma \vdash exec(c\text{'}c') \Leftrightarrow (exec(c) \wedge \\ \bigvee_{c_i} c' = c_i \ \wedge \ c\text{'}cond_\pi(c') \ \wedge \ \bigwedge_{c_r} \neg exec(c\text{'}c_r)) \end{array}}$$

P2X

$$\frac{\begin{array}{c} c_1 \rightarrow_\pi c_2 \\ \pi, \Gamma \vdash exec(\varsigma\text{'}c_1) \qquad \pi, \Gamma \vdash exec(\varsigma\text{'}c_2) \end{array}}{\pi, \Gamma \vdash \varsigma\text{'}c_1 \prec \varsigma\text{'}c_2}$$

OX2IX

$$\frac{\begin{array}{c} \pi, \Gamma \vdash c_1 \prec c_2 \\ \pi, \Gamma \vdash exec(c_1\text{'}c_3) \qquad \pi, \Gamma \vdash exec(c_2\text{'}c_4) \end{array}}{\pi, \Gamma \vdash c_1\text{'}c_3 \prec c_2\text{'}c_4}$$

TSEQ

$$\frac{\begin{array}{c} \pi, \Gamma \vdash exec(l_1) \qquad \pi, \Gamma \vdash exec(l_2) \\ \pi, \Gamma \vdash thread(l_1) = thread(l_2) \\ \pi, \Gamma \vdash obj(l_1) = obj(l_2) = \textbf{this} \ \vee \\ (\neg obj(l_1) = \textbf{this} \ \wedge \ \neg obj(l_2) = \textbf{this}) \end{array}}{\pi, \Gamma \vdash l_1 \prec l_2 \ \vee \ l_2 \prec l_1 \ \vee \ l_1 = l_2}$$

CALLER

$$\frac{\begin{array}{c} tpar_\pi(n) = t \qquad par1_\pi(n) = x \\ Returns_\pi(n) = \{\overline{c_i}\} \\ \pi, \Gamma \vdash exec(c) \\ \pi, \Gamma \vdash obj(c) = \textbf{this} \ \wedge \ name(c) = n \end{array}}{\begin{array}{c} \pi, \Gamma \vdash c\text{'}t = thread(c) \ \wedge \ c\text{'}x = arg(c) \wedge \\ \bigvee_{i=1..n} exec(c\text{'}c_i) \ \wedge \ arg1(c\text{'}c_i) = retv(c) \end{array}}$$

CALLEE

$$\frac{\begin{array}{c} tpar_\pi(n) = t \qquad par1_\pi(n) = x \\ c' \in Labels_\pi(n) \\ \pi, \Gamma \vdash exec(\varsigma\text{'}c') \end{array}}{\begin{array}{c} \pi, \Gamma \vdash \neg(\varsigma = \epsilon) \ \wedge \ exec(\varsigma) \wedge \\ obj(\varsigma) = \textbf{this} \ \wedge \ name(\varsigma) = n \\ thread(\varsigma) = \varsigma\text{'}t \ \wedge \ arg(\varsigma) = \varsigma\text{'}x \end{array}}$$

RET

$$\frac{\begin{array}{c} tpar_\pi(n) = t \qquad par1_\pi(n) = x \\ c' \in Returns_\pi(n) \\ \pi, \Gamma \vdash exec(c\text{'}c') \end{array}}{\begin{array}{c} \pi, \Gamma \vdash exec(c) \wedge \\ obj(c) = \textbf{this} \ \wedge \ name(c) = n \wedge \\ thread(c) = c\text{'}t \ \wedge \ arg(c) = c\text{'}x \wedge \\ retv(c) = arg1(c\text{'}c') \end{array}}$$

TLOCAL

$$\frac{\begin{array}{c} \mathcal{T}(basename(o)) = ThreadLocal \ st \\ \pi, \Gamma \vdash exec(l_1) \ \wedge \ exec(l_2) \\ \pi, \Gamma \vdash obj(l_1) = obj(l_2) = o \end{array}}{\pi, \Gamma \vdash thread(l_1) = thread(l_2)}$$

TREAL

$$\frac{\begin{array}{c} \pi, \Gamma \vdash \tau \prec\!\!\prec \tau' \\ \pi, \Gamma \vdash exec(l) \ \wedge \ thread(l) = \tau \\ \pi, \Gamma \vdash exec(l') \ \wedge \ thread(l') = \tau' \end{array}}{\pi, \Gamma \vdash l \prec l'}$$

Figure 5.9: Structure Inference Rules. All of the rules have the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

$$\text{IX2OX}\quad \frac{\pi, \Gamma \vdash c_1\text{'}c_3 \prec c_2\text{'}c_4 \qquad \pi, \Gamma \vdash thread(c_1\text{'}c_3) = thread(c_2\text{'}c_4)}{\pi, \Gamma \vdash c_1 \prec c_2 \ \vee \ c_1 = c_2}$$

Figure 5.10: Derived Structure Inference Rules

another *this* method call c_2, then every executed method call of the body of c_1 is executed before every executed method call of the body of c_2.

The rule TSeq states that every thread is sequential. Every two *this* method calls by the same thread are ordered in the execution order. Similarly, every two method calls on base objects by the same thread are ordered in the execution order.

The rule Caller states that if a *this* method call is executed, its parameters and arguments are equal and that one of the return statements in its body is executed and its return value is equal to the value that the executed return statement returns.

The rule Callee states that if a method call in the body of a *this* method call is executed, then the *this* method call is executed and the parameters and the arguments of the *this* method call are equal.

The rule Ret states that if a return statement of the body of a *this* method call is executed, then the *this* method call is executed and the parameters and the arguments of the *this* method call are equal and the return value of the *this* method call is the value that the return statement returns.

The rule TLocal states that every two executed method calls on the same thread-local object are from the same thread.

The rule TReal states that if a thread is ordered before another thread, then every method call from the former is executed before every method call from the latter.

The rule IX2OX states that if two method calls in the body of two *this* method calls execute in order by the same thread, then the two *this* method calls execute in the same

order.

5.5.3 Basic Inference Rules

The basic inference rules axiomatize the properties of the execution and linearization orders and their interdependence. The basic inference rules state are presented in 5.11. The derived basic inference rules are presented in Figure 5.12. We explain each rule in turn.

The rule XASym states the asymmetry property of the execution order. If a method call is executed before another method call, then the latter is not executed before the former and they are not executed concurrently.

The rule XTrans states the transitivity property of the precedence execution order. The rule XXTrans states the transitivity of the sequence of precedence, concurrency and precedence execution relations. If l_1 is executed before l_2, l_2 is executed (before or) concurrent to l_3 and l_3 is executed before l_4, then l_1 is executed before l_4.

The rule XTotal states the totality property of the precedence and concurrency execution relations. Every two method calls either execute in order or concurrently.

The rule X2X states that if a method call is executed before another one, then obviously both are executed.

The rule X2L states the real-time-preservation property of linearization orders. The execution order of two method calls on a linearizable object is preserved in the linearization order.

The rule LASym states the asymmetry property of linearization orders. If a method call is linearized before another one, then the latter is not linearized before the former.

The rule LTrans states the transitivity property of linearization orders.

The rule LTotal states the totality property of linearization orders.

The rule L2X states that if a method call is linearized before another one, then obviously both are executed.

The rule P2L states that the program order of two method calls on a linearizable object

is preserved in the linearization order.

The rule XLTrans is a form of "transitivity" rule for judgments about the execution order $\prec$ and the linearization order $\prec_o$ for a linearizable object o. If l_1 is executed before l_2, l_2 is linearized before l_3 and l_3 is executed before l_4, then l_1 is executed before l_4.

The rule X2L' states the contra-positive of the rule X2L.

XASym
$$\frac{\pi, \Gamma \vdash l \prec l'}{\pi, \Gamma \vdash \neg(l' \prec l) \ \wedge \ \neg(l' \sim l) \ \wedge \ \neg(l' = l)}$$

XTrans
$$\frac{\pi, \Gamma \vdash l \prec l' \qquad \pi, \Gamma \vdash l' \prec l''}{\pi, \Gamma \vdash l \prec l''}$$

XXTrans
$$\frac{\pi, \Gamma \vdash l_1 \prec l_2 \qquad \pi, \Gamma \vdash l_3 \prec l_4 \qquad \pi, \Gamma \vdash l_2 \sim l_3}{\pi, \Gamma \vdash l_1 \prec l_4}$$

XTotal
$$\frac{\pi, \Gamma \vdash exec(l) \ \wedge \ exec(l')}{\pi, \Gamma \vdash (l \prec l') \ \vee \ (l' \prec l) \ \vee \ (l \sim l') \ \vee \ (l = l')}$$

X2X
$$\frac{\pi, \Gamma \vdash l \prec l'}{\pi, \Gamma \vdash exec(l) \ \wedge \ exec(l')}$$

X2L
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi, \Gamma \vdash obj(l) = obj(l') = o \qquad \pi, \Gamma \vdash l \prec l'}{\pi, \Gamma \vdash l \prec_o l'}$$

LASym
$$\frac{\pi, \Gamma \vdash l \prec_o l'}{\pi, \Gamma \vdash \neg(l' \prec_o l) \ \wedge \ \neg(l = l')}$$

LTrans
$$\frac{\pi, \Gamma \vdash l \prec_o l' \qquad \pi, \Gamma \vdash l' \prec_o l''}{\pi, \Gamma \vdash l \prec_o l''}$$

LTotal
$$\frac{\mathcal{T}_{base}(o) \in LT \cup SCT \qquad \pi, \Gamma \vdash exec(l) \ \wedge \ exec(l') \qquad \pi, \Gamma \vdash obj(l) = obj(l') = o}{\pi, \Gamma \vdash (l \prec_o l') \ \vee \ (l' \prec_o l) \ \vee \ (l = l')}$$

L2X
$$\frac{\mathcal{T}_{base}(o) \in LT \cup SCT \qquad \pi, \Gamma \vdash l \prec_o l'}{\pi, \Gamma \vdash exec(l) \ \wedge \ exec(l') \wedge \ obj(l) = obj(l') = o}$$

All of the rules have the side condition $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Figure 5.11: Basic Inference Rules

P2L
$$\frac{c_1 \rightarrow_\pi c_2 \qquad \pi, \Gamma \vdash exec(\varsigma' c_1) \qquad \pi, \Gamma \vdash exec(\varsigma' c_2) \qquad \mathcal{T}_{base}(o) \in LT \qquad \pi, \Gamma \vdash obj(\varsigma' c_1) = obj(\varsigma' c_2) = o}{\pi, \Gamma \vdash \varsigma' c_1 \prec_o \varsigma' c_2}$$

XLTrans
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi, \Gamma \vdash l_1 \prec l_2 \qquad \pi, \Gamma \vdash l_3 \prec l_4 \qquad \pi, \Gamma \vdash l_2 \prec_o l_3}{\pi, \Gamma \vdash l_1 \prec l_4}$$

X2L'
$$\frac{\mathcal{T}_{base}(o) \in LT \qquad \pi, \Gamma \vdash l \prec_o l'}{\pi, \Gamma \vdash l \precsim l'}$$

Figure 5.12: Derived Basic Inference Rules

5.5.4 Synchronization Object Inference Rules

The synchronization object inference rules axiomatize the properties of common synchronization object types. We consider each type in turn.

Basic and Atomic Register. The basic and atomic register inference rules are presented in Figure 5.13.

The rule AReg states that for every read method call l_R on an atomic register, there is a write method call ℓ_W on it that writes the same value that l_R returns and ℓ_W is the last write method call that is linearized before l_R.

A method call $\not{l}$ is race-free $isRaceFree_r(\not{l})$ if an only if there is no write method call that executes concurrent to it. A register reg is sequentially-written $isSequentiallyWritten(reg)$ if and only if every pair of write method calls on it are ordered in the execution order or in other words, every write method call on it is race-free.

The rule BReg states that if a basic register reg is sequentially-written, for every race-free read method call l_R on reg, there is a write method call ℓ_W on reg that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R. Note that this models Lamport's notion of safe registers [48].

The derived register inference rules are presented in Figure 5.14.

The rule AReg' states that for every read method call l_R on an atomic register, if l_W is the last write method call that is linearized before l_R, then l_W writes the same value that l_R returns.

An object o is accessed sequentially $isSequential(o)$ if and only if every pair of method calls on it are ordered in the execution order.

The rule BReg' states that if a basic register reg is accessed sequentially, for every read method call l_R on reg, there is a write method call ℓ_W on reg that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R.

The rule TReg states that for every read method call l_R on a thread-local register reg,

there is a write method call ℓ_W on reg that writes the same value that l_R returns and ℓ_W is the last write method call that is executed before l_R.

$$\textsc{AReg}\quad \frac{\mathcal{T}_{base}(reg) = AtomicRegister \qquad \pi, \Gamma \vdash isRead_{reg}(l_R)}{\pi, \Gamma \vdash \exists \ell_W : isWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

$$\textsc{BReg}\quad \frac{\mathcal{T}_{base}(reg) = BasicRegister \qquad \pi, \Gamma \vdash isSequentiallyWritten(reg) \qquad \pi, \Gamma \vdash isRead_{reg}(l_R) \qquad \pi, \Gamma \vdash isRaceFree_{reg}(l_R)}{\pi, \Gamma \vdash \exists \ell_W : isEWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

$$\begin{aligned}
&isRead_r(\mathit{l}_R) \Leftrightarrow \\
&\quad exec(\mathit{l}_R) \ \wedge \ obj(\mathit{l}_R) = r \ \wedge \ name(\mathit{l}_R) = read \\
&isWrite_r(\mathit{l}_W) \Leftrightarrow \\
&\quad exec(\mathit{l}_W) \ \wedge \ obj(\mathit{l}_W) = r \ \wedge \ name(\mathit{l}_W) = write \\
&isWriter_r(\mathit{l}_W, \mathit{l}_R) \Leftrightarrow \\
&\quad isWrite_r(\mathit{l}_W) \ \wedge \ \mathit{l}_W \prec_r \mathit{l}_R \wedge \\
&\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq_r \mathit{l}_W \ \vee \ \mathit{l}_R \prec_r \ell'_W) \\
&isEWriter_r(\mathit{l}_W, \mathit{l}_R) \Leftrightarrow \\
&\quad isWrite_r(\mathit{l}_W) \ \wedge \ \mathit{l}_W \prec \mathit{l}_R \wedge \\
&\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq \mathit{l}_W \ \vee \ \mathit{l}_R \prec \ell'_W) \\
&isSequential(o) \Leftrightarrow \\
&\quad \forall \ell, \ell' : (exec(\ell) \ \wedge \ exec(\ell') \ \wedge \ obj(\ell) = o \ \wedge \ obj(\ell') = o) \Rightarrow \\
&\qquad (\ell \preceq \ell' \ \vee \ \ell' \prec \ell) \\
&isRaceFree_r(\mathit{l}) \Leftrightarrow \\
&\quad \forall \ell_W : isWrite_r(\ell_W) \Rightarrow (\ell_W \prec \mathit{l} \ \vee \ \mathit{l} \prec \ell_W) \\
&isSequentiallyWritten(r) \Leftrightarrow \\
&\quad \forall \ell_w : isWrite_r(\ell_w) \Rightarrow isRaceFree_r(\ell_w)
\end{aligned}$$

Figure 5.13: Register Inference Rules.

AReg'

$$\frac{\mathcal{T}_{base}(reg) = AtomicRegister \quad \pi, \Gamma \vdash isRead_{reg}(l_R) \quad \pi, \Gamma \vdash isWriter_{reg}(l_W, l_R)}{\pi, \Gamma \vdash arg1(l_W) = retv(l_R)}$$

BReg'

$$\frac{\mathcal{T}_{base}(reg) = BasicRegister \quad \pi, \Gamma \vdash isRead_{reg}(l_R) \quad \pi, \Gamma \vdash isSequential(reg)}{\pi, \Gamma \vdash \exists \ell_W : isEWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

TReg

$$\frac{\mathcal{T}(reg) = ThreadLocal\ BasicRegister \quad \pi, \Gamma \vdash isRead_{reg[\tau]}(l_R)}{\pi, \Gamma \vdash \exists \ell_W : isEWriter_{reg[\tau]}(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W) \wedge thread(\ell_W) = \tau}$$

Figure 5.14: Derived Register Inference Rules

CAS Atomic Register. The cas register inference rules are presented in Figure 5.15.

A method call $\not{l}_W$ on an atomic cas register r is a successful write $isCWrite_r(\not{l}_W)$, if and only if it is a write method call or a successful cas method call. The written value $writtenValue(\not{l})$ of a successful write method call $\not{l}$ is its first argument if it is a write method call and its second argument if it a successful cas method call.

The rule CASRegRead states that for every read method call l_R on an atomic cas register, there is a successful write ℓ_W that writes the same value that l_R has returned and ℓ_W is the last successful write that is linearized before l_R.

The rule CASRegCAST and the rule CASRegCASF state that a cas method call l_C on an atomic cas register returns *true* if the written value of the last successful write linearized before l_C is equal to the first argument of l_C, and returns $false$ otherwise.

The derived cas register inference rules are presented in Figure 5.16.

The rule CASRegRead' states that for every read method call l_R on an atomic cas register, the last successful write that is linearized before l_R writes the same value that l_R returns.

CASRegRead

$$\frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \quad \pi, \Gamma \vdash isRead_{reg}(l_R)}{\pi, \Gamma \vdash \exists \ell_W : isCWriter_{reg}(\ell_W, l_R) \wedge retv(l_R) = writtenValue(\ell_W)}$$

CASRegCAST

$$\frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \quad \pi, \Gamma \vdash isCAS_{reg}(l_C) \quad \pi, \Gamma \vdash isCWriter_{reg}(l_W, l_R) \quad \pi, \Gamma \vdash arg1(l_C) = writtenValue(l_W)}{\pi, \Gamma \vdash retv(l_C) = true}$$

CASRegCASF

$$\frac{\mathcal{T}_{base}(reg) = AtomicCASRegister \quad \pi, \Gamma \vdash isCAS_{reg}(l_C) \quad \pi, \Gamma \vdash isCWriter_{reg}(l_W, l_R) \quad \pi, \Gamma \vdash \neg(arg1(l_C) = writtenValue(l_W))}{\pi, \Gamma \vdash retv(l_C) = false}$$

$$
\begin{aligned}
&isRead_r(\mathit{l}_R) \Leftrightarrow \\
&\quad exec(\mathit{l}_R) \ \wedge \ obj(\mathit{l}_R) = r \ \wedge \ name(\mathit{l}_R) = read \\
&isWrite_r(\mathit{l}_R) \Leftrightarrow \\
&\quad exec(\mathit{l}_R) \ \wedge \ obj(\mathit{l}_R) = r \ \wedge \ name(\mathit{l}_R) = write \\
&isCAS_r(\mathit{l}_R) \Leftrightarrow \\
&\quad exec(\mathit{l}_R) \ \wedge \ obj(\mathit{l}_R) = r \ \wedge \ name(\mathit{l}_R) = cas \\
&isCWrite_r(\mathit{l}_W) \Leftrightarrow \\
&\quad isWrite_r(\mathit{l}_W) \ \vee \ (isCAS_r(\mathit{l}_W) \ \wedge \ retv(\mathit{l}_W) = true) \\
&isCWriter_r(\mathit{l}_W, \mathit{l}_R) \Leftrightarrow \\
&\quad isCWrite_r(\mathit{l}_W) \ \wedge \ \mathit{l}_W \prec_r \mathit{l}_R \wedge \\
&\quad \forall \ell'_W : isCWrite_r(\mathit{l}'_W) \Rightarrow (\ell'_W \preceq_r \mathit{l}_W \ \vee \ \mathit{l}_R \prec_r \ell'_W) \\
&writtenValue(\mathit{l}) = \\
&\quad \begin{cases} arg1(\mathit{l}) & \text{if } obj(\mathit{l}) = write \\ arg2(\mathit{l}) & \text{if } obj(\mathit{l}) = cas \end{cases}
\end{aligned}
$$

Figure 5.15: CAS Register Inference Rules.

CASRegRead'

$$\frac{\mathcal{T}_{base}(reg) = CASAtomicRegister \quad \pi, \Gamma \vdash isRead_{reg}(l_R) \quad \pi, \Gamma \vdash isCWriter_{reg}(l_W, l_R)}{\pi, \Gamma \vdash retv(l_R) = writtenValue(l_W)}$$

Figure 5.16: Derived CAS Register Inference Rules

Lock and Try-Lock. The preliminary definitions are presented in Figure 5.17 and the lock and try-lock inference rules are presented in Figure 5.18.

Ownership for a lock l is respected, $isOwnerRespecting(l)$ if and only if every thread unlocks l only if it has already locked l and has not unlocked it since then.

The rule LOCK states that if ownership is respected for a lock l and a *lock* method call on l (by a thread t_1) is linearized before an *unlock* method call on l (by a thread t_2), then an *unlock* method call on l by t_1 is linearized before a *lock* method call on l by t_2.

The rule LOCKREADL states that if ownership is respected for a lock l and an *unlock* method call on l (by a thread t) is linearized after a *read* method call on l that returns $false$, then a *lock* method call on l by t is linearized after the *read* method call.

The rule LOCKREADR states that if ownership is respected for a lock l and a *lock* method call on l (by a thread t) is linearized before a *read* method call on l that returns $false$, then an *unlock* method call on l by t is linearized before the *read* method call.

The rule LOCKREADM states that if ownership is respected for a lock l and a *read* method call on l (by a thread t) is linearized between a pair of matching *lock* and *unlock* method call on l, then the read method call returns $true$.

There are four similar rules for try-locks. Instead of *lock* method calls, these rules concern successful lock method calls that are *lock* and successful *tryLock* method calls.

$$
\begin{aligned}
&isLock_o(\mathit{l}) \Leftrightarrow \\
&\quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = lock \\
&isUnlock_o(\mathit{l}) \Leftrightarrow \\
&\quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = unlock \\
&isRead_o(\mathit{l}) \Leftrightarrow \\
&\quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = read \\
&isTryLock_o(\mathit{l}) \Leftrightarrow \\
&\quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = tryLock \\
&isTLock_o(\mathit{l}) \Leftrightarrow \\
&\quad isLock_o(\mathit{l}) \ \vee \ (isTryLock_o(\mathit{l}) \ \wedge \ retv(\mathit{l}) = true) \\
&noUnlockBetween_o(\mathit{l}_l, \mathit{l}_u) \Leftrightarrow \\
&\quad \forall \ell'_u : \\
&\qquad (isXUnlock_{X,o}(\ell'_u) \wedge \\
&\qquad thread_X(\mathit{l}_l) = thread_X(\ell'_u)) \Rightarrow \\
&\qquad (\ell'_u \prec \mathit{l}_l \ \vee \ \mathit{l}_u \preceq \ell'_u)
\end{aligned}
$$

$$
\begin{aligned}
&isOwnerRespecting(o) \Leftrightarrow \\
&\quad \forall \ell : isUnlock_o(\ell) \Rightarrow \\
&\quad \exists \ell' : isTLock_o(\ell') \wedge \\
&\qquad thread(\ell') = thread(\ell) \wedge \\
&\qquad \ell' \prec \ell \wedge \\
&\qquad \forall \ell'' : \\
&\qquad\quad (isUnLock_o(\ell'') \wedge \\
&\qquad\quad thread(\ell'') = thread(\ell)) \\
&\qquad \Rightarrow \\
&\qquad\quad \ell'' \prec \ell' \ \vee \ \ell \preceq \ell''
\end{aligned}
$$

Figure 5.17: Preliminary definitions for Lock and TryLock Inference Rules.

LOCK

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isLock_o(l_{l_1}) \qquad \pi, \Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \ \wedge \\ isLock_o(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \ \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

TRYLOCK

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isTLock_o(l_{l_1}) \qquad \pi, \Gamma \vdash isUnlock_o(l_{u_2}) \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{u_2}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{u_1}, \ell_{l_2}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \ \wedge \\ isTLock_o(\ell_{l_2}) \ \wedge \ thread(\ell_{l_2}) = thread(l_{u_2}) \ \wedge \\ \ell_{u_1} \prec_o \ell_{l_2}\end{array}}$$

LOCKREADL

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi, \Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{l_1}: \\ isLock_o(\ell_{l_1}) \ \wedge \ thread(\ell_{l_1}) = thread(l_{u_1}) \ \wedge \\ l_{r_2} \prec_o \ell_{l_1}\end{array}}$$

TRYLOCKREADL

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi, \Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{l_1}: \\ isTLock_o(\ell_{l_1}) \ \wedge \ thread(\ell_{l_1}) = thread(l_{u_1}) \ \wedge \\ l_{r_2} \prec_o \ell_{l_1}\end{array}}$$

LOCKREADR

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isLock_o(l_{l_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{r_2}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{u_1}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \ \wedge \\ \ell_{u_1} \prec_o l_{r_2}\end{array}}$$

TRYLOCKREADR

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isTLock_o(l_{l_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \ \wedge \ retv(l_{r_2}) = false \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{r_2}\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_{u_1}: \\ isUnlock_o(\ell_{u_1}) \ \wedge \ thread(\ell_{u_1}) = thread(l_{l_1}) \ \wedge \\ \ell_{u_1} \prec_o l_{r_2}\end{array}}$$

LOCKREADM

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = Lock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isLock_o(l_{l_1}) \qquad \pi, \Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi, \Gamma \vdash thread(l_{u_1}) = thread(l_{l_1}) \\ \pi, \Gamma \vdash noUnlockBetween_o(l_{l_1}, l_{u_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{r_2} \qquad \pi, \Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\pi, \Gamma \vdash retv(l_{r_2}) = true}$$

TRYLOCKREADM

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = TryLock \\ \pi, \Gamma \vdash isOwnerRespecting(o) \\ \pi, \Gamma \vdash isTLock_o(l_{l_1}) \qquad \pi, \Gamma \vdash isUnlock_o(l_{u_1}) \\ \pi, \Gamma \vdash thread(l_{u_1}) = thread(l_{l_1}) \\ \pi, \Gamma \vdash noUnlockBetween_o(l_{l_1}, l_{u_1}) \\ \pi, \Gamma \vdash isRead(l_{r_2}) \\ \pi, \Gamma \vdash l_{l_1} \prec_o l_{r_2} \qquad \pi, \Gamma \vdash l_{r_2} \prec_o l_{u_1}\end{array}}{\pi, \Gamma \vdash retv(l_{r_2}) = true}$$

Figure 5.18: Lock and TryLock Inference Rules.

Strong Counter. The strong counter inference rules are presented in Figures 5.19 and 5.20.

The rule SCounter states that the return value of every method call that is linearized before an *iaf* method call is smaller than the return value of the *iaf* method call.

The rule SCounter' states that if the return value of a method call is greater than the return value of an *iaf* method call, then it is linearized after the *iaf* method call.

SCounter

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = SCounter \\ \pi, \Gamma \vdash obj(l_1) = o \\ \pi, \Gamma \vdash obj(l_2) = o \ \wedge \ name(l_2) = iaf \\ \pi, \Gamma \vdash l_1 \prec_o l_2\end{array}}{\pi, \Gamma \vdash retv(l_1) < retv(l_2)}$$

Figure 5.19: SCounter Rules

SCounter'

$$\frac{\begin{array}{c}\mathcal{T}_{base}(o) = SCounter \\ \pi, \Gamma \vdash exec(l_1) \ \wedge \ obj(l_1) = o \\ \pi, \Gamma \vdash exec(l_2) \ \wedge \ obj(l_2) = o \ \wedge \ name(l_2) = iaf \\ \pi, \Gamma \vdash retv(l_1) > retv(l_2)\end{array}}{\pi, \Gamma \vdash l_2 \prec_o l_1}$$

Figure 5.20: Derived SCounter Rules

Basic Set and Basic Map. The Set and Map inference rules are presented in Figure 5.21.

An object o is accessed sequentially $isSequential(o)$ if and only if every pair of method calls on it are ordered in the execution order.

The rule BasicSetContains states that if a basic set s is accessed sequentially, for every *contains* method call on s that returns *true*, there is a preceding *add* method call on s with the same argument.

The rule BasicSetAdd states that if a basic set s is accessed sequentially, every *contains* method call on s that succeeds an *add* method call on s with the same argument returns *true*.

The rule BasicMapGet states that if a basic map m is accessed sequentially, for every *get* method call l_g on m that does not return $\perp$, there exists a *put* method call ℓ_p on m with the same key argument such that the value argument of p is equal to the return value of l_g and ℓ_p is the latest preceding *put* method call on m with the same key argument.

The rule BasicMapPut states that if a basic map m is accessed sequentially, for every *get* method call l_g on m, if l_p is the latest preceding *put* method call on m with the same key argument then the value argument of l_p is equal to the return value of l_g.

The derived Set and Map inference rules are presented in Figure 5.22.

The rule BasicMapGet' states that if a basic map m is accessed sequentially, for every *get* method call l_g on m, if no *put* method call with the same key argument as l_g precedes l_g, then l_g returns $\perp$.

The rule BasicMapPut' states that if a basic map m is accessed sequentially and no *put* method call puts $\perp$ in m, every *get* method call that succeeds a *put* method call with the same key argument does not return $\perp$.

BasicSetContains

$$\frac{\begin{array}{c}\mathcal{T}_{base}(s) = BasicSet \\ \pi, \Gamma \vdash isSequential(s) \\ \pi, \Gamma \vdash isContains_s(l_c) \ \wedge \ retv(l_c) = true\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_a \colon isAdd_s(\ell_a) \wedge \\ arg1(\ell_a) = arg1(l_c) \ \wedge \ \ell_a \prec l_c\end{array}}$$

BasicSetAdd

$$\frac{\begin{array}{c}\mathcal{T}_{base}(s) = BasicSet \\ \pi, \Gamma \vdash isSequential(s) \\ \pi, \Gamma \vdash isAdd_s(l_a) \\ \pi, \Gamma \vdash isContains_s(l_c) \\ \pi, \Gamma \vdash l_a \prec l_c \ \wedge \ arg1(l_a) = arg1(l_c)\end{array}}{\pi, \Gamma \vdash retv(l_c) = true}$$

BasicMapGet

$$\frac{\begin{array}{c}\mathcal{T}_{base}(m) = BasicMap \\ \pi, \Gamma \vdash isSequential(m) \\ \pi, \Gamma \vdash isGet_m(l_g) \ \wedge \ retv(l_g) \neq \bot\end{array}}{\begin{array}{c}\pi, \Gamma \vdash \exists \ell_p \colon isPutter_m(\ell_p, l_g) \wedge \\ arg2(\ell_p) = retv(l_g)\end{array}}$$

BasicMapPut

$$\frac{\begin{array}{c}\mathcal{T}_{base}(m) = BasicMap \\ \pi, \Gamma \vdash isSequential(m) \\ \pi, \Gamma \vdash isGet_m(l_g) \\ \pi, \Gamma \vdash isPutter_m(l_p, l_g)\end{array}}{\pi, \Gamma \vdash arg2(l_p) = retv(l_g)}$$

$$\begin{array}{l} isContains_o(\mathit{l}) \Leftrightarrow \\ \quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = contains \\ isAdd_o(\mathit{l}) \Leftrightarrow \\ \quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = add \\ isPut_o(\mathit{l}) \Leftrightarrow \\ \quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = put \\ isGet_o(\mathit{l}) \Leftrightarrow \\ \quad exec(\mathit{l}) \ \wedge \ obj(\mathit{l}) = o \ \wedge \ name(\mathit{l}) = get \\ isPutter_m(\mathit{l}_p, \mathit{l}_g) \Leftrightarrow \\ \quad isPut_m(\mathit{l}_p) \ \wedge \ arg1(\mathit{l}_p) = arg1(\mathit{l}_g) \ \wedge \ \mathit{l}_p \prec \mathit{l}_g \ \wedge \\ \quad \forall \ell'_p \colon isPut_m(\ell'_p) \ \wedge \ arg1(\ell'_p) = arg1(\mathit{l}_g) \Rightarrow (\ell'_p \preceq \mathit{l}_p \ \vee \ \mathit{l}_g \prec \ell'_p) \\ isSequential(o) \Leftrightarrow \\ \quad \forall \ell, \ell' \colon (exec(\ell) \ \wedge \ exec(\ell') \ \wedge \ obj(\ell) = o \ \wedge \ obj(\ell') = o) \Rightarrow \\ \qquad (\ell \preceq \ell' \ \vee \ \ell' \prec \ell) \end{array}$$

Figure 5.21: Set and Map Inference Rules

$$\textsc{BasicMapGet'}\quad \frac{\begin{array}{c}\mathcal{T}_{base}(m) = BasicMap \\ \pi, \Gamma \vdash isSequential(m) \\ \pi, \Gamma \vdash isGet_m(l_g) \\ \pi, \Gamma \vdash \neg\exists \ell_p \colon isPut_m(\ell_p) \wedge \\ arg1(\ell_p) = arg1(l_g) \ \wedge \ \ell_p \prec l_g\end{array}}{\pi, \Gamma \vdash retv(l_g) = \bot}$$

$$\textsc{BasicMapPut'}\quad \frac{\begin{array}{c}\mathcal{T}_{base}(m) = BasicMap \\ \pi, \Gamma \vdash isSequential(m) \\ \pi, \Gamma \vdash isGet_m(l_g) \\ \pi, \Gamma \vdash isPut_m(l_p) \\ \pi, \Gamma \vdash arg1(l_p) = arg1(l_g) \ \wedge \ l_p \prec l_g \\ \pi, \Gamma \vdash \forall \ell_p \colon isPut_m(\ell_p) \Rightarrow arg2(\ell_p) \neq \bot\end{array}}{\pi, \Gamma \vdash retv(l_g) \neq \bot}$$

Figure 5.22: Derived Set and Map Inference Rules

5.6 Soundness

In this section, we present the soundness, exchange, and weakening lemmas for SOL.

The semantics satisfies the classical exchange and weakening lemmas.

Lemma 36 (Exchange)**.**

$\forall \pi, \Gamma, \Gamma', \mathcal{A}, \mathcal{A}', \mathcal{A}'':$

$(\pi, \Gamma; \mathcal{A}; \mathcal{A}'; \Gamma' \vdash \mathcal{A}'') \Rightarrow (\pi, \Gamma; \mathcal{A}'; \mathcal{A}; \Gamma' \vdash \mathcal{A}'')$

Lemma 37 (Weakening)**.**

$\forall \pi, \Gamma, \mathcal{A}, \mathcal{A}':$

$(\pi, \Gamma \vdash \mathcal{A}) \Rightarrow (\pi, \Gamma; \mathcal{A}' \vdash \mathcal{A})$

To define the soundness, we first define the models relation between specifications and assertions.

Definition 16. *A specification π models an assertion $\mathcal{A}$ if and only if every execution of π models $\mathcal{A}$.*

$\pi \models \mathcal{A}$ *iff* $\forall \mathcal{X} \in [\![\pi]\!] : \mathcal{X} \models \mathcal{A}$

The logic is sound. The following theorem states that the logic derives valid conclusions from valid premises.

Theorem 3 (Soundness)**.**

$\forall \pi, \mathcal{A}: ((\pi, \Gamma \vdash \mathcal{A}) \ \wedge \ (\pi \models \Gamma)) \Rightarrow (\pi \models \mathcal{A})$.

See the appendix section 10.4 for the proof.

5.7 Dekker Mutual Exclusion

In this section, we prove the mutual exclusion guarantee of the Dekker algorithm using SOL.

We presented the Dekker algorithm, π_{Dekker}, in Figure 2.1.

Theorem 4 (Mutual Exclusion)**.**

In every execution of the Dekker specification, at most one thread acquires the lock.

$\forall X \in \mathbb{H}(\pi_{Dekker})$: $(retv_X(L_2) = true) \Rightarrow (retv_X(L_1) = false)$.

Proof.

We show that

(1) $X' \in \mathbb{H}(\pi_{Dekker})$

We show that

(2) $(retv_{X'}(L_2) = true) \Rightarrow$
$(retv_{X'}(L_1) = false)$

By Definition 2.71 on [1], we have that there exists $\mathcal{X}, X, \sigma, \mathcal{L}$ such that

(3) $\mathcal{X} = (X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

(4) $X' = \sigma(X)$

By Lemma 38, we have

(5) $\pi_{Dekker}, \cdot \vdash (retv(L_2) = true) \Rightarrow$
$(retv(L_1) = false)$.

By the soundness theorem, Theorem 3, and Definition [16] on [5] and [3], we have

(6) $\mathcal{X} \models (retv(L_2) = true) \Rightarrow$
$(retv(L_1) = false)$

By Definition [15] on [6], [3] and [4], we have

(7) $(retv_{X'}(L_2) = true) \Rightarrow$

$(retv_{X'}(L_1) = false)$. □

Lemma 38.

$\pi_{Dekker}, \cdot \vdash (retv(L_2) = true) \Rightarrow (retv(L_1) = false)$.

Proof.

Let

$\pi = \pi_{Dekker}$

We show that

$\pi, \cdot \vdash (retv(L_2) = true) \Rightarrow$

$(retv(L_1) = false)$

Let

(8) $\Gamma = (retv(L_2) = true)$

By rule CondIntro, we have to show that

$\pi, \Gamma \vdash retv(L_1) = false$

By rule Premise on [8], we have

(9) $\pi, \Gamma \vdash retv(L_2) = true$

From π, we have

$cond_\pi(L_2) = true$

Thus,

(10) $\pi, \Gamma \vdash cond_\pi(L_2) = true$

By rule OControl on [10], we have

(11) $\pi, \Gamma \vdash exec(L_2)$

From π, we have

(12) $name_\pi(L_2) = tryLock2$

(13) $R_1 \in Labels(tryLock2)$

From π, we have

$cond_\pi(R_1) = true$

Thus,

(14) $\pi, \Gamma \vdash L_2\text{'}cond_\pi(R_1) = true$

From π, we have

(15) $PreReturns_\pi(R_1) = \emptyset$

By rule ICONTROL on [11]-[15], we have

(16) $\pi, \Gamma \vdash exec(L_2\text{'}R_1)$

By rule ID on [16], we have

(17) $\pi, \Gamma \vdash obj(L_2\text{'}R_1) = f_1$

(18) $\pi, \Gamma \vdash name(L_2\text{'}R_1) = read$

(19) $\pi, \Gamma \vdash retv(L_2\text{'}R_1) = L_2\text{'}x_1$

Similarly, we have

(20) $\pi, \Gamma \vdash exec(L_2\text{'}W_2)$

(21) $\pi, \Gamma \vdash obj(L_2\text{'}W_2) = f_2$

(22) $\pi, \Gamma \vdash name(L_2\text{'}W_2) = write$

(23) $\pi, \Gamma \vdash arg1(L_2\text{'}W_2) = 1$

From the definition of $isRead$ on [16], [17] and [18] and rule CONJINTRO, we have

(24) $\pi, \Gamma \vdash isRead_{f_1}(L_2\text{'}R_1)$

From rule AREG on [24], we have

(25) $\pi, \Gamma \vdash \exists \ell_W :$

$isWriter_{f_1}(\ell_W, L_2\text{'}R_1) \wedge$

$retv(L_2\text{'}R_1) = arg1(\ell_W)$

Let

(26) $\Gamma' = \Gamma;$

$isWriter_{f_1}(l_W, L_2\text{'}R_1) \wedge$

$arg1(l_W) = retv(L_2\text{'}R_1)$

where l_W is fresh.

By rule PREMISE on [26], we have

(27) $\pi, \Gamma' \vdash isWriter_{f_1}(l_W, L_2\text{'}R_1)$

(28) $\pi, \Gamma' \vdash arg1(l_W) = retv(L_2\text{'}R_1)$

By rule ID on [11], we have

(29) $\pi, \Gamma \vdash obj(L_2) = this$

(30) $\pi, \Gamma \vdash name(L_2) = tryLock2$

From π, we have

(31) $Returns_\pi(tryLock2) = \{C_{2t}, C_{2f}\}$

By rule CALLER on [31], [11], [30], [31], we have

(32) $\pi, \Gamma \vdash$

$(exec(L_2\text{'}C_{2t}) \ \wedge \ arg1(L_2\text{'}C_{2t}) = retv(L_2)) \vee$

$(exec(L_2\text{'}C_{2f}) \ \wedge \ arg1(L_2\text{'}C_{2f}) = retv(L_2))$

We apply rule DISJELIM to [32]:

Right:

Let

(33) $\Gamma' = \Gamma;$

$(exec(L_2\text{'}C_{2f}) \ \wedge \ arg1(L_2\text{'}C_{2f}) = retv(L_2))$

By rule PREMISE on [33], we have

(34) $\pi, \Gamma' \vdash exec(L_2\text{'}C_{2f})$

(35) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2f}) = retv(L_2)$

From π, we have

(36) $arg1(C_{2f}) = false$

By rule ID on [34], [36], we have

(37) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2f}) = false$

From rule ETRANS and rule ESYM on [35], and [37], we have

(38) $\pi, \Gamma' \vdash retv(L_2) = false$

By weakening (Lemma 37) on [33] [9], we have

(39) $\pi, \Gamma' \vdash retv(L_2) = true$

By rule NegElim on [38] and [39], we have

(40) $\pi, \Gamma' \vdash retv(L_1) = false$

Left:

Let

(41) $\Gamma' = \Gamma$;

$(exec(L_2\text{'}C_{2t}) \ \wedge \ arg1(L_2\text{'}C_{2t}) = retv(L_2))$

By rule Premise on [41], we have

(42) $\pi, \Gamma' \vdash exec(L_2\text{'}C_{2t})$

(43) $\pi, \Gamma' \vdash arg1(L_2\text{'}C_{2t}) = retv(L_2)$

From π, we have

(44) $cond_\pi(C_{2t}) = (x_1 = 0)$

By rule IControl on [42] and [44] we have

(45) $\pi, \Gamma' \vdash (L_2\text{'}x_1 = 0)$

From [28], [19], [45], weakening (Lemma 37) and rule ETrans, we have

(46) $\pi, \Gamma' \vdash arg1(l_W) = 0$

From the definition of $isWriter$ on [27] and rule ConjElimL and rule ConjElimR, we have

(47) $\pi, \Gamma' \vdash obj(l_W) = f_1$

(48) $\pi, \Gamma' \vdash name(l_W) = write$

(49) $\pi, \Gamma' \vdash exec(l_W)$

(50) $\pi, \Gamma' \vdash l_W \prec_{f_1} L_2\text{'}R_1$

(51) $\pi, \Gamma' \vdash \forall \ell_{W'} : isWrite_{f_1}(\ell_{W'}) \Rightarrow$

$\ell_{W'} \preceq_{f_1} l_W \ \vee \ L_2\text{'}R_1 \prec_{f_1} \ell_{W'}$

From the definition of π, we have

(52) $calls_\pi(f_1, write) = \{W_1, W_{01}\}$

From rule Src on [47], [48], [49] and [52], we have that for some fresh ς

(53) $\pi, \Gamma' \vdash l_W = \varsigma\text{'}W_1 \ \vee \ l_W = \varsigma\text{'}W_{01}$

We apply rule DisjElim to [53]:

Left:

(54) $\Gamma'' = \Gamma'$;

$l_W = \varsigma\text{'}W_1$

From [49], [54], weakening (Lemma 37), we have

(55) $\pi, \Gamma'' \vdash exec(\varsigma\text{'}W_1)$

From π, we have

(56) $arg1_\pi(W_1) = 1$

By rule Id on [54], [56], we have

(58) $\pi, \Gamma'' \vdash arg1(\varsigma\text{'}W_1) = 1$

From [54], [58], we have

(58) $\pi, \Gamma'' \vdash arg1(l_W) = 1$

By weakening (Lemma 37) on [46], we have

(59) $\pi, \Gamma'' \vdash arg1(l_W) = 0$

By rule ETrans and rule ESym on [58], [59], we have

(60) $\pi, \Gamma'' \vdash 0 = 1$

By rule NegElim on rule Zero and [60], we have

(61) $\pi, \Gamma'' \vdash retv(L_1) = false$

Right:

(62) $\Gamma'' = \Gamma'$;

$l_W = \varsigma\text{'}W_{01}$

By rule Premise on [62], we have

(63) $\pi, \Gamma'' \vdash l_W = \varsigma\text{'}W_{01}$

From π, we have

(64) $W_{01} \in Labels_\pi(init)$

By rule CALLEE on [63] and [64] we have

(65) $\pi, \Gamma'' \vdash \neg(\varsigma = \epsilon)$

(66) $\pi, \Gamma'' \vdash exec(\varsigma)$

(67) $\pi, \Gamma'' \vdash obj(\varsigma) = this$

(68) $\pi, \Gamma'' \vdash name(\varsigma) = init$

From π, we have

(69) $Calls_\pi(this, init) = \{L_0\}$

By rule SRC on [65]-[69] we have

(70) $\pi, \Gamma'' \vdash \varsigma = L_0$

From [63] and [70], we have

(71) $\pi, \Gamma'' \vdash l_W = L_0\text{'}W_{01}$

From π, we have

$cond_\pi(L_1) = true$

Thus,

(72) $\pi, \Gamma'' \vdash cond_\pi(L_1) = true$

By rule OCONTROL on [72], we have

(73) $\pi, \Gamma'' \vdash exec(L_1)$

From π, we have

(74) $name_\pi(L_1) = tryLock1$

(75) $R_2 \in Labels(tryLock1)$

From π, we have

$cond_\pi(R_2) = true$

Thus,

(76) $\pi, \Gamma'' \vdash L_1\text{'}cond_\pi(R_2) = true$

From π, we have

(77) $PreReturns_{\pi}(R_2) = \emptyset$

By rule IControl on [73]-[77], we have

(78) $\pi, \Gamma'' \vdash exec(L_1\text{'}R_2)$

From π we have

(79) $obj_{\pi}(R_2) = f_2$

(80) $name_{\pi}(R_2) = read$

(81) $retv_{\pi}(R_2) = x_2$

By rule Id on [78] and [79]-[81], and then rule ConjElimL and rule ConjElimR, we have

(82) $\pi, \Gamma'' \vdash obj(L_1\text{'}R_2) = f_2$

(83) $\pi, \Gamma'' \vdash name(L_1\text{'}R_2) = read$

(84) $\pi, \Gamma'' \vdash retv(L_1\text{'}R_2) = L_1\text{'}x_2$

From the definition of $isRead$ on [78], [82], [83] and rule ConjIntro, we have

(85) $\pi, \Gamma'' \vdash isRead_{f_2}(L_1\text{'}R_2)$

Similarly, we have that

(86) $\pi, \Gamma'' \vdash exec(L_1\text{'}W_1)$

(87) $\pi, \Gamma'' \vdash obj(L_1\text{'}W_1) = f_1$

(88) $\pi, \Gamma'' \vdash name(L_1\text{'}W_1) = write$

(89) $\pi, \Gamma'' \vdash arg1(L_1\text{'}W_1) = 1$

(90) $\pi, \Gamma'' \vdash isWrite_{f_1}(L_1\text{'}W_1)$

By rule UnivElim on [51], and [90], we have

(91) $\pi, \Gamma'' \vdash L_1\text{'}W_1 \preceq_{f_1} l_W \vee \ L_2\text{'}R_1 \prec_{f_1} L_1\text{'}W_1$

By rule LSubs on [91] and [71], we have

(92) $\pi, \Gamma'' \vdash$

$L_1\text{'}W_1 \preceq_{f_1} L_0\text{'}W_{01} \vee$

$L_2\text{'}R_1 \prec_{f_1} L_1\text{'}W_1$

From π, we have

(93) $L_0 \rightarrow_\pi L_1$

By rule LSubs on [66] and [70], we have

(94) $\pi, \Gamma'' \vdash exec(L_0)$

By rule P2X on [93], [94] and [73], we have

(95) $\pi, \Gamma'' \vdash L_0 \prec L_1$

By rule LSubs on [49] and [71], we have

(96) $\pi, \Gamma'' exec(L_0{}'W_{01})$

By rule OX2IX on [95], and [96], and [86], we have

(97) $\pi, \Gamma'' \vdash L_0{}'W_{01} \prec L_1{}'W_1$

By rule Id on [96], we have

(98) $\pi, \Gamma'' \vdash obj(L_0{}'W_{01}) = f_1$

By rule X2L on [97], [98] and [87], we have

(99) $\pi, \Gamma'' \vdash L_0{}'W_{01} \prec_{f_1} L_1{}'W_1$

By rule LASym on [99], and rule ConjElimL, we have

(100) $\pi, \Gamma'' \vdash \neg(L_1{}'W_1 \prec_{f_1} L_0{}'W_{01})$

By rule DisjSyllL on [92], [100], we have

(101) $\pi, \Gamma'' \vdash L_2{}'R_1 \prec_{f_1} L_1{}'W_1$

From π, we have

(102) $W_2 \rightarrow_\pi R_1$

From rule P2X on [102], [20], [16], and weakening (Lemma 37), we have

(103) $\pi, \Gamma'' \vdash L_2{}'W_2 \prec L_2{}'R_1$

From π, we have

(104) $W_1 \rightarrow_\pi R_2$

From rule P2X on [104], [86] and [78], we have

(105) $\pi, \Gamma'' \vdash L_1{}'W_1 \prec L_1{}'R_2$

From rule XLTrans on [103], [101] and [105], we have

(106) $\pi, \Gamma'' \vdash L_2{}'W_2 \prec L_1{}'R_2$

From rule X2L on [106], [21] and [82], we have

(107) $\pi, \Gamma'' \vdash L_2\text{'}W_2 \prec_{f_2} L_1\text{'}R_2$

We show that

(108) $\pi, \Gamma'' \vdash \forall \ell_W :$

$$isWrite_{f_2}(\ell_W) \Rightarrow$$

$$\ell_W \preceq_{f_2} L_2\text{'}W_2 \ \vee \ L_1\text{'}R_2 \prec_{f_2} \ell_W$$

Let

(109) $\Gamma''' = \Gamma''; isWrite_{f_2}(l'_W)$

By rule UnivIntro and rule CondIntro,

we have to show that

$$\pi, \Gamma''' \vdash l'_W \preceq_{f_2} L_2\text{'}W_2 \ \vee \ L_1\text{'}R_2 \prec_{f_2} l'_W$$

By rule Premise on [109], we have

(110) $\pi, \Gamma''' \vdash isWrite_{f_2}(l'_W)$

From definition of $isWrite$ on [110],

we have

(111) $\pi, \Gamma''' \vdash$

$$obj(l'_W) = f_2 \wedge$$

$$name(l'_W) = write \wedge$$

$$exec(l'_W)$$

From the definition of π, we have

(112) $calls_\pi(f_2, write) = \{W_{02}, W_2\}$

By rule Src on [111] and [112], we have that

for some fresh ς,

(113) $\pi, \Gamma''' \vdash l'_W = \varsigma\text{'}W_{02} \ \vee \ l'_W = \varsigma\text{'}W_2$

We apply rule DisjElim on [113]:

Left:

(114) $\Gamma'''' = \Gamma'''; (l'_W = \varsigma\text{'}W_{02})$

By rule PREMISE on [114], we have

(115) $\pi, \Gamma'''' \vdash l'_W = \varsigma\text{'}W_{02}$

By rule LSUBS on [111], [115] and

weakening (Lemma 37), we have

(116) $\pi, \Gamma'''' \vdash exec(\varsigma\text{'}W_{02})$

From π, we have

(117) $W_{02} \in Labels_\pi(init)$

By rule CALLEE on [116] and [117], we have

(118) $\pi, \Gamma'''' \vdash \neg(\varsigma = \epsilon)$

(119) $\pi, \Gamma'''' \vdash exec(\varsigma)$

(120) $\pi, \Gamma'''' \vdash obj(\varsigma) = this$

(121) $\pi, \Gamma'''' \vdash name(\varsigma) = init$

From π, we have

(122) $calls_\pi(this, init) = \{L_0\}$

By rule SRC on [118]-[122], we have

(123) $\pi, \Gamma'''' \vdash \varsigma = L_0$

By rule LSUBS on [115], [123], we have

(124) $\pi, \Gamma'''' \vdash l'_W = L_0\text{'}W_{02}$

By rule LSUBS on [111], [124], we have

(125) $\pi, \Gamma''' \vdash obj(L_0\text{'}W_{02}) = f_2$

(126) $\pi, \Gamma''' \vdash exec(L_0\text{'}W_{02})$

From π, we have

(127) $L_0 \rightarrow_\pi L_2$

By rule P2X on [127], [94] and [11],

weakening (Lemma 37), we have

(128) $\pi, \Gamma'''' \vdash L_0 \prec L_2$

By rule OX2IX on [128], and [126], and [20],

we have

(129) $\pi, \Gamma'''' \vdash L_0\text{'}W_{02} \prec L_2\text{'}W_2$

By rule X2L on [129], and [125], and [21],

we have

(130) $\pi, \Gamma'''' \vdash L_0\text{'}W_{02} \prec_{f_2} L_2\text{'}W_2$

By rule DisjIntroL on [130], we have

(131) $\pi, \Gamma''' \vdash$

$$L_0\text{'}W_{02} \preceq_{f_2} L_2\text{'}W_2 \vee$$

$$L_1\text{'}R_2 \prec_{f_2} L_0\text{'}W_{02}$$

By rule LSubs on [131] and [124], we have

(132) $\pi, \Gamma''' \vdash$

$$l'_W \preceq_{f_2} L_2\text{'}W_2 \vee$$

$$L_1\text{'}R_2 \prec_{f_2} l'_W$$

Right:

(133) $\Gamma'''' = \Gamma'''; (l'_W = \varsigma\text{'}W_2)$

By rule Premise on [133], we have

(134) $\pi, \Gamma'''' \vdash l'_W = \varsigma\text{'}W_2$

Similar to the previous part, we can show

that

(135) $\pi, \Gamma'''' \vdash \varsigma = L_2$

By rule LSubs on [134] and [135], we have

(136) $\pi, \Gamma'''' \vdash l'_W = L_2\text{'}W_2$

By rule DisjIntroR on [136], we have

(137) $\pi, \Gamma''' \vdash l'_W \preceq_{f_2} L_2\text{'}W_2$

Thus, by rule DisjIntroL on [137], we

have

$\pi, \Gamma''' \vdash$

$l'_W \preceq_{f_2} L_2\text{'}W_2 \vee$

$L_1\text{'}R_2 \prec_{f_2} l'_W$

By rule ConjIntro and the definition of $isWrite$ on [20]-[22] and weakening (Lemma 37), we have

(138) $\pi, \Gamma'' \vdash isWrite_{f_2}(L_2\text{'}W_2)$

By rule ConjIntro and the definition of $isWriter$ on [138], [107], and [108], we have

(139) $\pi, \Gamma'' \vdash isWriter_{f_2}(L_2\text{'}W_2, L_1\text{'}R_2)$

By rule ConjIntro and the definition of $isRead$ on [78], [82] and [83], we have

(140) $\pi, \Gamma'' \vdash isRead_{f_2}(L_1\text{'}R_2)$

From rule AReg' on [140] and [139], we have

(141) $\pi, \Gamma'' \vdash retv(L_1\text{'}R_2) = arg1(L_2\text{'}W_2)$

By rule ETrans and rule ESym on [141], [84] and [23], we have

(142) $\pi, \Gamma'' \vdash L_1\text{'}x_2 = 1$

By rule Zero and rule ESubs on [142], we have

(143) $\pi, \Gamma'' \vdash \neg(L_1\text{'}x_2 = 0)$

From π, we have that

(144) $cond_\pi(C_{1f}) = \neg(x_2 = 0)$

(145) $name_\pi(L_1) = tryLock1$

(146) $C_{1f} \in Labels_\pi(tryLock1)$

(147) $PreReturns_\pi(C_{1f}) = \emptyset$

From [144], we have

(148) $L_1\text{'}cond_\pi(C_{1f}) = \neg(L_1\text{'}x_2 = 0)$

From [143] and [148], we have

(149) $\pi, \Gamma'' \vdash L_1\text{'}cond_\pi(C_{1f})$

By rule IControl on [73], [146], [145], [149], [147] we have

(150) $\pi, \Gamma'' \vdash exec(L_1\text{'}C_{1f})$

From π, we have that

(151) $C_{1f} \in Returns_\pi(tryLock1)$

(152) $arg1_\pi(C_{1f}) = false$

By rule Id on [150] and [152], we have

(153) $\pi, \Gamma'' \vdash arg1(L_1\text{'}C_{1f}) = false$

By rule Ret on [150], [151], we have

(154) $\pi, \Gamma'' \vdash retv(L_1) = arg1(L_1\text{'}C_{1f})$

By rule ETrans and rule ESym on [153], [154], we have

$\pi, \Gamma'' \vdash retv(L_1) = false$

□

Chapter 6

Syntactic TM Correctness

We define the correctness of TM algorithms as a *syntactic property* of them. The syntactic statement of the correctness condition in the program logic makes it possible to verify a TM algorithm specification by syntactic deductions instead of model checking all execution histories of the specification.

In this chapter, we first axiomatize the properties of the client transactions and prove that they are valid for every TM algorithm specification.

Then, we define the markability assertions as the correctness condition of TM algorithm specifications. The markability assertions are parametrized with the marking relation. The effect and access orders of a TM algorithm specification should be captured as its marking relation. We say that a TM algorithm specification is markable if there is a marking of it such that assuming the client transaction axioms, the markability assertions can be derived. We prove that a TM algorithm specification is opaque if it is markable. Therefore, defining the marking relation and then deriving the markability assertions is a sound proof technique for opacity of TM algorithm specifications.

6.1 Client Transactions

In the subsection 2.2.2, we defined client transactions. In this subsection, we axiomatize the properties of the client transactions as a set of assertions and prove their validity for every TM algorithm specification. These properties are later assumed to prove markability.

Let us define

$$isInit(\ell) = exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = init \tag{6.1}$$

$$isRead(\ell) = exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = read \tag{6.2}$$

$$isWrite(\ell) = exec(l) \wedge obj(\ell) = this \wedge name(\ell) = write \tag{6.3}$$

$$isCommit(\ell) = exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = commit \tag{6.4}$$

$$isCommitted(\tau) = \exists \ell \colon isCommit(\ell) \wedge thread(\ell) = \tau \wedge retv(\ell) = \mathbb{C} \tag{6.5}$$

$$isAborted(\tau) = \exists \ell \colon exec(\ell) \wedge obj(\ell) = this \wedge thread(\ell) = \tau \wedge retv(\ell) = \mathbb{A} \tag{6.6}$$

Transactions have the following set of properties. Γ_1: Every transaction is initialized. Γ_2: Every transaction is initialized only once. Γ_3: The initialization operation of each transaction is executed before its other operations. Γ_4: If a transaction is committed, it executed the commit operation. Γ_5: Every transaction executes the commit operation at most once. Γ_6: The commit operation of each transaction is executed after its other operations. Γ_7: Each

transaction is either aborted or committed.

$$\Gamma_0 = \Gamma_1 \wedge \Gamma_2 \wedge \Gamma_3 \wedge \Gamma_4 \wedge \Gamma_5 \wedge \Gamma_6 \wedge \Gamma_7 \tag{6.7}$$

$$\Gamma_1 = \forall t\colon \mathit{Let}\ l = \mathit{initOf}(t)\colon \mathit{isInit}(l) \wedge \mathit{thread}(l) = t \tag{6.8}$$

$$\Gamma_2 = \forall \ell, \ell'\colon (\mathit{isInit}(\ell) \wedge \mathit{isInit}(\ell') \wedge \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell = \ell' \tag{6.9}$$

$$\Gamma_3 = \forall \ell, \ell'\colon (\mathit{isInit}(\ell) \wedge \mathit{exec}(\ell') \wedge \mathit{obj}(\ell') = \mathit{this} \wedge \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell \preceq \ell' \tag{6.10}$$

$$\Gamma_4 = \forall t\colon \mathit{Let}\ l = \mathit{commitOf}(t)\colon \mathit{isCommitted}(t) \Rightarrow (\mathit{isCommit}(l) \wedge \mathit{thread}(l) = t) \tag{6.11}$$

$$\Gamma_5 = \forall \ell, \ell'\colon (\mathit{isCommit}(\ell) \wedge \mathit{isCommit}(\ell') \wedge \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell = \ell' \tag{6.12}$$

$$\Gamma_6 = \forall \ell, \ell'\colon (\mathit{exec}(\ell) \wedge \mathit{obj}(\ell) = \mathit{this} \wedge \mathit{isCommit}(\ell') \wedge \mathit{thread}(\ell) = \mathit{thread}(\ell')) \Rightarrow \ell \preceq \ell' \tag{6.13}$$

$$\Gamma_7 = \forall t\colon (\mathit{isCommitted}(t) \wedge \neg \mathit{isAborted}(t)) \vee (\mathit{isAborted}(t) \wedge \neg \mathit{isCommitted}(t)) \tag{6.14}$$

The following lemma states that these properties of client transactions are valid for every TM algorithm specification.

Lemma 39. $\forall \pi \in \Pi_{TM}\colon \pi \models \Gamma_0$.

See the appendix section 10.5.1 for the proof.

The following lemma states that if an assertion $\mathcal{A}$ is derived for a TM specification π assuming the properties of the client transactions, then $\mathcal{A}$ is valid for π.

Lemma 40. $\forall \pi \in \Pi_{TM}, \forall \mathcal{A}\colon (\pi, \Gamma_0 \vdash \mathcal{A}) \Rightarrow (\pi \models \mathcal{A})$.

See the appendix section 10.5.1 for the proof.

$$isTRead(\lfloor_R) \Leftrightarrow \tag{6.15}$$
$$\quad isRead(\lfloor_R) \ \wedge \ retv(\lfloor_R) \neq \mathbb{A}$$
$$isLocalTRead(\lfloor_R) \Leftrightarrow \tag{6.16}$$
$$\quad isTRead(\lfloor_R) \wedge$$
$$\quad \exists \ell_W : isTWrite(\ell_W) \ \wedge \ arg1(\lfloor_R) = arg1(\ell_W) \ \wedge \ thread(\lfloor_R) = thread(\ell_W) \ \wedge \ \ell_W \prec \lfloor_R$$
$$isGlobalTRead(\lfloor_R) = \tag{6.17}$$
$$\quad isTRead(\lfloor_R) \ \wedge \ \neg isLocalTRead(\lfloor_R)$$
$$isTWrite(\lfloor_W) = \tag{6.18}$$
$$\quad isWrite(\lfloor_W) \ \wedge \ retv(\lfloor_W) \neq \mathbb{A}$$
$$isLocalTWrite(\lfloor_W) = \tag{6.19}$$
$$\quad isTWrite(\lfloor_W) \wedge$$
$$\quad \exists \ell'_W : isTWrite(\ell'_W) \ \wedge \ arg1(\lfloor_W) = arg1(\ell'_W) \ \wedge \ thread(\lfloor_W) = thread(\ell'_W) \ \wedge \ \lfloor_W \prec \ell'_W$$
$$isGlobalTWrite(\lfloor_W) = \tag{6.20}$$
$$\quad isTWrite(\lfloor_W) \ \wedge \ \neg isLocalTWrite(\lfloor_W)$$
$$isCommitted(\tau) = \tag{6.21}$$
$$\quad \exists \ell : exec(\ell) \ \wedge \ obj(\ell) = this \ \wedge \ thread(\ell) = \tau \ \wedge \ retv(\ell) = \mathbb{C}$$
$$isTWriter_i(\tau) = \tag{6.22}$$
$$\quad \exists \ell_W : isTWrite(\ell_W) \ \wedge \ arg1(\ell_W) = i \ \wedge \ thread(\ell_W) = \tau \ \wedge \ isCommitted(\tau)$$

Figure 6.1: Reads and Writes

6.2 Markability

Some preliminary definitions are presented in Figure 6.1. The marking assertions are defined in Figure 6.2. These program assertions mirror history assertions defined in Figure 3.4 and 3.5. We illustrated the notion of markability in the subsection 3.3.

We define the set of Markable TM algorithm as follows:

Definition 17 (Markable TM Algorithm)**.** *A TM algorithm π is markable, if and only if there exists a marking relation $\sqsubseteq$ such that assuming Γ_0, $isMarking(\sqsubseteq)$ can be derived for π.*

$Markable = \{\pi \mid \exists \sqsubseteq : \ \pi, \Gamma_0 \vdash isMarking(\sqsubseteq)\}$.

To prove the markability of a TM algorithm specification, its access and effect orders

$$isMarkingRel(\sqsubseteq) \Leftrightarrow \tag{6.23}$$
$$\forall t_1, t_2, t_3 :$$
$$(t_1 \sqsubseteq t_2 \ \vee \ t_2 \sqsubseteq t_1) \wedge$$
$$(t_1 \sqsubseteq t_2 \ \wedge \ t_2 \sqsubseteq t_1) \Rightarrow (t_1 = t_2) \wedge$$
$$(t_1 \sqsubseteq t_2) \ \wedge \ (t_2 \sqsubseteq t_3) \Rightarrow (t_1 \sqsubseteq t_3) \wedge$$
$$\forall \ell_R, t : \mathit{Let}\ i = arg1(\ell_R) :$$
$$(isGlobalTRead(\ell_R) \ \wedge \ isTWriters_i(t)) \Rightarrow$$
$$(\ell_R \sqsubseteq t \ \vee \ t \sqsubseteq \ell_R) \wedge$$
$$(\ell_R \sqsubseteq t \Rightarrow \neg t \sqsubseteq \ell_R) \ \wedge \ (t \sqsubseteq \ell_R \Rightarrow \neg \ell_R \sqsubseteq t)$$

$$NoWriteBetween_{t,i}(\not{l}, \not{l}') \Leftrightarrow \tag{6.24}$$
$$\forall \ell : (isTWrite(\ell) \ \wedge \ thread(\ell) = t \ \wedge \ arg1(\ell) = i) \Rightarrow (\ell \preceq \not{l} \ \vee \ \not{l}' \preceq \ell)$$

$$NoWriterBetween_i(q_1, \sqsubseteq, q_2) \Leftrightarrow \tag{6.25}$$
$$\forall t : isTWriter_i(t) \Rightarrow t \sqsubseteq q_1 \ \vee \ q_2 \sqsubseteq t$$

$$isLocalWriteObs \Leftrightarrow \tag{6.26}$$
$$\forall \ell_R : isLocalTRead(\ell_R) \Rightarrow \mathit{Let}\ t = thread(\ell_R), i = arg1(\ell_R) :$$
$$\exists \ell_W : isTWrite(\ell_W) \ \wedge \ thread(\ell_W) = t \ \wedge \ arg1(\ell_W) = i \wedge$$
$$\ell_w \prec \ell_R \ \wedge \ NoWriteBetween_{t,i}(\ell_W, \ell_R) \wedge$$
$$retv(\ell_R) = arg2(\ell_W)$$

$$isLastPreAccessor_{\sqsubseteq}(t', \not{l}_R) \Leftrightarrow \tag{6.27}$$
$$\mathit{Let}\ i = arg1(\not{l}_R), t = thread(\not{l}_R) :$$
$$isTWriter_i(t') \wedge$$
$$t' \sqsubseteq \not{l}_R \ \wedge \ t' \neq t \wedge$$
$$NoWriterBetween_i(t', \not{l}_R)$$

$$isGlobalWriteObs(\sqsubseteq) \Leftrightarrow \tag{6.28}$$
$$\forall \ell_R : isGlobalTRead(\ell_R) \Rightarrow \exists \ell_W : isGlobalTWrite(\ell_W) \ \wedge \ \mathit{Let}\ t' = thread(\ell_W) :$$
$$isLastPreAccessor_{\sqsubseteq}(t', \ell_R) \wedge$$
$$arg1(\ell_R) = arg1(\ell_W) \ \wedge \ retv(\ell_R) = arg2(\ell_W)$$

$$isWriteObs(\sqsubseteq) \Leftrightarrow \tag{6.29}$$
$$isLocalWriteObs \ \wedge \ isGlobalWriteObs(\sqsubseteq)$$

$$isReadPres(\sqsubseteq) \Leftrightarrow \tag{6.30}$$
$$\forall \ell_R : isGlobalTRead(\ell_R) \Rightarrow \mathit{Let}\ i = arg1(\ell_R), t = thread(\ell_R) :$$
$$NoWriterBetween_i(\ell_R, \sqsubseteq, t) \ \wedge \ NoWriterBetween_i(t, \sqsubseteq, \ell_R)$$

$$isRealTimePres(\sqsubseteq) \Leftrightarrow \tag{6.31}$$
$$\forall t, t' : t \not\preceq t' \Rightarrow t \sqsubseteq t'$$

$$isMarking(\sqsubseteq) \Leftrightarrow \tag{6.32}$$
$$isMarkingRel(\sqsubseteq) \ \wedge \ isWriteObs(\sqsubseteq) \ \wedge \ isReadPres(\sqsubseteq) \ \wedge \ isRealTimePres(\sqsubseteq)$$

Figure 6.2: $isMarking$ Assertions

should be captured as the marking relation and assuming the client transaction axioms, the markability assertions should be derived for the specification. Markability is a proof technique for opacity. The following theorem states the soundness of this technique. A TM algorithm is opaque if the markability assertion is derivable for its specification.

Theorem 5 (Markability Soundness)**.** *A TM algorithm is opaque if it is markable.*
$Markable \subseteq Opaque.$

See the appendix section 10.5.2 for the proof.

Chapter 7

Verification of TM Algorithms

In the previous chapter, we presented markability as a proof technique for opacity of TM algorithm specifications. In this chapter, we prove the markability of two TM algorithms.

Given a TM algorithm specification, the access and effect orders of the algorithm can be readily *captured* as the execution and linearization order of specific method calls in the specification. The markability assertions can, then, be proved using the inference rules of the logic. First, we look at TL2 algorithm and then DSTM (visible reads) algorithm.

7.1 Marking TL2

Now, we define the marking relation for the TL2 algorithm specification that is presented in Figure 2.2.

Let us define

$$Eff(\tau) = \begin{cases} initOf(\tau)\text{'}I01 & \text{if } isAborted(\tau) \\ commitOf(\tau)\text{'}C07 & \text{if } isCommitted(\tau) \end{cases} \tag{7.1}$$

$$readAcc(l_R) = l_R\text{'}R04 \tag{7.2}$$

$$writeAcc_i(\tau) = commitOf(\tau)\text{'}C16_i \tag{7.3}$$

The marking $\sqsubseteq$ is the reflexive closure of $\sqsubset$ that is define as follows:

$$
\begin{array}{l}
\forall t, t' : \\
\quad t \sqsubset t' \Leftrightarrow Eff(t) \prec_{clock} Eff(t') \\
\forall \ell_R, t : isTRead(\ell_R) \ \wedge \ isTWriter_i(t) \Rightarrow \\
\quad Let\ i = arg1(\ell_R) : \\
\quad t \sqsubset \ell_R \Leftrightarrow writeAcc_i(t) \precsim readAcc(\ell_R) \\
\quad \ell_R \sqsubset t \Leftrightarrow readAcc(\ell_R) \prec writeAcc_i(t)
\end{array}
\tag{7.4}
$$

The effect order of transactions is the linearization order of their calls to the *clock*. The *clock* object numbers the snapshots. Every transaction reads an initial snapshot number at $I01$. A committing transaction makes a new snapshot at $C08$. A TL2 transaction takes effect at $I01$ if it is aborted and at $C08$ if it is committed.

The access order of read operations and writer transactions to location i is the execution order of their access to the $reg[i]$ register. The read method reads $reg[i]$ at $R04$ and a writer transaction writes to $reg[i]$ at $C16_i$.

The following lemma states that the relation $\sqsubseteq$ defined above is a marking relation for π_{TL2}.

Lemma 41. $\pi_{TL2}, \Gamma_0 \vdash isMarking(\sqsubseteq)$.

This lemma is fully proved using the program logic formalized in PVS. The proof scripts are available at [49].

TL2 maintains write-observation by acquiring locks for the written locations in the commit method and also the orders $R03 \rightarrow R04$, $R05 \rightarrow R06$, $C16 \rightarrow C17 \rightarrow C18$ and checking that the lock is released and that the two versions are equal in the read method.

TL2 maintains read-preservation by validations in both the read and the commit methods. The read-preservation is maintained in the read method by the orders $R04 \rightarrow R05 \rightarrow R06$, $C17 \rightarrow C18$ and checking that the lock is released and the read version is no larger than the initial snapshot in the read method. The read-preservation is similarly maintained in the

commit method by the orders $C7 \rightarrow C10 \rightarrow C11$, $C17 \rightarrow C18$ and checking that the lock is released and the read version is no larger than the initial snapshot in the commit method. Checking that the lock is released forces concurrent writers to write their new versions so that the version check finds the violation.

Now, we can have opacity of TL2.

Theorem 6 (Opacity of TL2)**.** $\pi_{TL2} \in Opaque$

Proof. Immediate from Theorem 41, Definition 17, and Theorem 5. □

7.2 Marking DSTM (visible reads)

We now conjecture a marking relation for the DSTM algorithm (visible reads) specification presented in Figure 2.5.

Let us have the following preliminary definitions

$$\begin{aligned}
isFirstTWrite_\tau(\natural_W) &= isTWrite(\natural_W) \ \wedge \ thread(\natural_W) = \tau \ \wedge && (7.5)\\
&\quad \forall \ell'_W \colon (isTWrite(\natural_W) \ \wedge \ thread(\ell'_W) = \tau) \Rightarrow \natural_W \preceq \ell_W \\
isLastTRead_\tau(\natural_R) &= isTRead(\natural_R) \ \wedge \ thread(\natural_R) = \tau \ \wedge && (7.6)\\
&\quad \forall \ell'_R \colon (isTRead(\natural_R) \ \wedge \ thread(\ell'_R) = \tau) \Rightarrow \ell_R \preceq \natural_R \\
isCommit_\tau(\natural_C) &= isCommit(\natural_C) \ \wedge \ thread(\ell_R) = \tau && (7.7)\\
hasTRead(\tau) &= \exists \ell_R \colon isTRead(\ell_R) \ \wedge \ thread(\ell_R) = \tau && (7.8)\\
isEffOp_\tau(\natural) &= (isAborted(\tau) \ \wedge \ \neg hasRead(\tau) \ \wedge \ isInit_\tau(\natural)) \ \vee && (7.9)\\
&\quad (isAborted(\tau) \ \wedge \ hasRead(\tau) \ \wedge \ isLastTRead_\tau(\natural)) \ \vee \\
&\quad (isCommitted(\tau) \ \wedge \ isCommit_\tau(\natural))
\end{aligned}$$

The assertion $isFirstTWrite_\tau(\natural_W)$ states that $\natural_W$ is the first *write* method call of transaction τ. The assertion $isLastTRead_\tau(\natural_R)$ states that $\natural_R$ is the last *read* method call of

transaction τ. The assertion $isCommit_\tau(\not{l}_C)$ states that $\not{l}_C$ is the *commit* method call of transaction τ. The assertion $hasTRead(\tau)$ states that transaction τ has a read method call. The assertion $isEffOp_\tau(\not{l})$ states that the method call $\not{l}$ is the effect operation of transaction τ.

Let us define

$$Eff(\not{l}) = \begin{cases} \not{l}\text{'}C01 & \text{if } isCommitted(\tau) \\ \not{l}\text{'}R05 & \text{if } isAborted(\tau) \ \wedge \ hasTRead(\tau) \\ \not{l}\text{'}I01 & \text{if } isAborted(\tau) \ \wedge \ \neg hasTRead(\tau) \end{cases} \tag{7.10}$$

$$readAcc(\not{l}_R) = \not{l}_R\text{'}R05 \tag{7.11}$$

$$writeAcc(\not{l}_W) = \not{l}_W\text{'}W12 \tag{7.12}$$

The marking $\sqsubseteq$ is the reflexive closure of $\sqsubset$ that is define as follows:

$$\begin{aligned} &\forall t, t' : \\ &\quad t \sqsubset t' \Leftrightarrow \\ &\qquad \forall \ell, \ell' : (isEffOp_t(\ell) \ \wedge \ isEffOp_{t'}(\ell')) \Rightarrow \\ &\qquad inv(Eff(\ell)) \lhd inv(Eff(\ell')) \\ &\forall \ell_R, t : isTRead(\ell_R) \ \wedge \ isTWriter_i(t) \Rightarrow \\ &\quad t \sqsubset \ell_R \Leftrightarrow \\ &\qquad \forall \ell_W : isFirstTWrite_t(\ell_W) \Rightarrow \\ &\qquad writeAcc(\ell_W) \prec_{start[i]} readAcc(\ell_R) \\ &\quad \ell_R \sqsubset t \Leftrightarrow \\ &\qquad \forall \ell_W : isFirstTWrite_t(\ell_W) \Rightarrow \\ &\qquad readAcc(\ell_R) \prec_{start[i]} writeAcc(\ell_W) \end{aligned} \tag{7.13}$$

A committed transactions takes effect at $C01$ of its commit method call where its state is *cas*ed from $\mathbb{R}$ to $\mathbb{C}$. An aborted transaction that has a successful read operation takes effect

at $R05$ of its last successful read where the the locator is updated. An aborted transaction that has no successful read operation takes effect at $I01$ of its initialization operation.

The access order of read operations and writer transactions to location i is the linearization order of their *cas* calls, $R05$ and $W12$, to the $start[i]$ register.

The algorithm maintains write-observation by deciding the stable value of a location based on the state of its last writer transaction.

As we illustrate with two examples in Figure 2.5 below, the algorithm maintains read-preservation by aborting the reader set when the location is being written and aborting the last writer transaction when the location is being read. Assume that the two locations i_1 and i_2 with the initial value v_1 are consistent if and only if they are equal. Transaction T_1 reads and transaction T_2 updates the values of the two locations to v_2.

While a location is being read, the algorithm aborts the last writer transaction of the location. Consider the example in Figure 7.1(a). If T_2 is allowed to commit after $read_{T_1}(i_1)$ then T_1 can read new value of i_2. The old value of i_1 (the value v_1) and the new value of i_2 (the value v_2) are inconsistent with each other. To prevent T_1 from reading inconsistent data, either T_1 or T_2 should be aborted. During $read_{T_1}(i_1)$, the algorithm aborts the last writer transaction that is T_2.

While a location is being written, the algorithm aborts the reader set of the location. Consider the example in Figure 7.1(b). Again, transaction T_1 can read inconsistent values. During $write_{T_2}(i_1, v_2)$, the algorithm aborts the readers set of location i_1 that includes T_1. Thus, T_1 will not execute its second read. (Note that aborting the reader set can be postponed until before committing the writer transaction (T_2). This allows the reader transactions to have the chance of committing before the writer commits.)

T_1	T_2
	$write_{T_2}(i_1, v_2)$
	$write_{T_2}(i_2, v_2)$
$read_{T_1}(i_1)$:v_1	
	$commit_{T_2}()$
$read_{T_1}(i_2)$:v_2	

(a) Aborting the Last Writer

T_1	T_2
$read_{T_1}(i_1)$:v_1	
	$write_{T_2}(i_1, v_2)$
	$write_{T_2}(i_2, v_2)$
	$commit_{T_2}()$
$read_{T_1}(i_2)$:v_2	

(b) Aborting the Reader Set

Figure 7.1: DSTM (visible reads) Preserving Reads

7.3 Marking NORec

Now, we present an informal definition of the marking relation for the NoRec algorithm specification of Figure 2.7.

Definition 18 (Marking NoRec)**.** *Consider an execution history $H \in \mathbb{H}(NORec)$. Let*

$$
\begin{aligned}
REff(T) &= \textit{The last execution of } I01 \textit{ or } V05 \\
Eff(T) &= \begin{cases} REff(T) & \textit{if } T \in Aborted(H) \ \vee \ TWrites(H) = \emptyset \\ commitOf(T)\text{'}C04 & \textit{if } T \in Committed(H) \ \wedge \ TWrites(H) \neq \emptyset \end{cases} \\
readAcc(T, i) &= \begin{cases} R\text{'}R03 & \textit{if } REff(T) \prec_H R\text{'}R03 \\ \textit{Let } REff(T) = V\text{'}V05 \textit{ in } V\text{'}V03_i & \textit{if } R\text{'}R03 \prec_H REff(T) \end{cases} \\
writeAcc(T, i) &= commitOf(T)\text{'}C07_i
\end{aligned}
$$

The marking $\sqsubseteq$ for H is the reflexive closure of $\sqsubset$ that is define as follows:

$$\{(T, T') \mid T, T' \in Trans(H) \ \wedge \ Eff(T) \prec_{seqLock} Eff(T')\} \cup$$
$$\{(T, R) \mid \exists i \colon R \in TReads(H), i = arg1(R), T \in Writers_H(i) \wedge$$
$$writeAcc(T, i) \prec_H readAcc(T, i)\} \cup$$
$$\{(R, T) \mid \exists i \colon R \in TReads(H), i = arg1(R), T \in Writers_H(i) \wedge$$
$$readAcc(T, i) \prec_H writeAcc(T, i)\}$$

An aborted transaction or a read-only transaction takes effect at the last execution of $I01$ or $V05$. This method call reads that most recent snapshot value that the transaction is still consistent for. A committed transactions that has write method calls takes effect at $C04$.

The access point of a read method call is at $R03$ if the last recent snapshot is read before $R03$; otherwise, it is at $V03_i$ of the latest successful validate method call. The access point of a writer transaction to location i is at $C07_i$.

Chapter 8

Related Works

8.1 Verification of Transactional Memory

Researchers have proposed several correctness criteria for the correctness of TM algorithms such as opacity [28], VWC [44], and TMS1 and TMS2 [22]. Lesani et al. [51] proved that opacity is stronger than TMS1 and weaker than TMS2. Considering the promised safety properties, designing a correct TM algorithm is a formidable task. Thus, verification of TM algorithms has been a topic of recent attention.

Researchers have employed model checking, automatic invariant generation and theorem proving to verify the correctness of TM algorithms. Model checkers from Cohen et al. [11, 12], and Guerraoui et al. [29, 31, 30] are the pioneering approach to verification of TM. Subsequently, the same approach was taken by O'Leary et al. [62] and Baek et al. [4]. Model checking can automate the verification process but is either based on assumed properties about the TM algorithm or only scalable to a finite number of threads and locations or simplified algorithms. Later, Emmi et al. [24] tried to automatically infer invariants that are strong enough to entail the correctness criterion. Compliance of the algorithms with the specification can be easily checked if the proper invariants can be automatically generated. On the other hand, this work reported resorting to simplified algorithms due to scalability

issues. Later, Lesani et al. [50] presented a machine checked theorem proving framework and a full proof of NORec TM algorithm [16]. The framework can be employed to verify realistic algorithms but requires translation of the algorithm to a transition system and more importantly, the process is manual and involves coming up with non-trivial invariants. Singh [73] developed a runtime verification tool for TM algorithms. Although the tool is optimized with sound approximation techniques, the runtime overhead is still not negligible. In contrast to the previous works, we presented a program logic for static verification of TM algorithms. The logic is general and does not assume any specific property of the algorithms. The reasoning is carried out on the algorithm specification itself rather than its transition system. We applied the program logic to machine-checked verification of realistic TM algorithms.

Testing can increase the reliability of a TM algorithm. Manovit et al. [53] applied random testing to find bugs in the TCC TM system. Lourenco et al. [52] encountered several bugs during the porting of TL2 algorithm and presented traces that exhibit these bugs. They presented test programs that can produce the bug traces. Both of the above works arbitrarily execute a test program and check that the execution instance does not exhibit a bug. On the other hand, given a TM algorithm and a bug pattern, our testing approach constructs a trace of the algorithm in the bug pattern if the algorithm is prone to the bug pattern.

Now, we consider each of the previous works in more detail.

Cohen et al. [11] verified small instances of some simple TM algorithms directly using a model checker. Inspired by the notions of *conflict* by Scott [70], they defined *admissible interchanges* of events in a history that can transform a concurrent history to a justifying sequential history. Later, the method was extended [12] to support non-transactional accesses to memory. This approach is limited to finite instances of the algorithms, especially for more complex algorithms.

Tasiran [74] presented a decomposition of serializability for a specific class of algorithms. Then, he verified that a particular algorithm refines each condition separately. Similar to

our notion of marking, this approach decomposes the correctness condition but is limited to a particular class of algorithms.

Guerraoui et al. [29, 31] specified both the TM algorithm and the correctness condition as transition systems. Verifying the correctness of the TM algorithm reduces to deciding language inclusion of the former in the latter. Due to unbounded number of threads and locations, the transition systems have infinite states. They tackled the problem with a small-world theorem that states that every TM algorithm satisfying certain structural properties is correct if and only if it is correct for *two threads and two memory locations*. This result has an immediate practical implication: Verification of a TM transition system reduces to verification of small instances of it. To our knowledge, these structural properties have not been formally verified for any TM algorithm.

In a follow up research, Emmi et al. [24] proposed a method that in contrast to [29, 31] did not presume properties for TM algorithms. They rewrote the transition systems of the TM algorithms and strict serializability that were presented by previous work [31] as transition systems parametrized with the number of threads and locations. The product (or composition) of two parametrized systems is defined as a transition system that on each command, essentially transitions for both systems. Verification of the TM algorithm reduces to model-checking the following logical statement in the product transition system of the TM and strict serializability: for every state, for every action, if the guard of the TM transition system is satisfied then the guard of strict serializability is also satisfied. This essentially means that at each state, the TM transition system allows an action only if strict serializability allows it. To verify that a target statement is an invariant of a system, ideas from verification by invisible invariants and template-based invariant generation are adapted. The verification procedure tries to come up with inductive invariants of the system and check if those invariants entail the target statement. To find an inductive invariant, candidate invariants are generated from a template schema. Small instances of the parametrized system are thoroughly generated and the candidate invariants that are not invariants of these instances

are filtered. The candidate inductive invariant is the conjunction of a subset of the remained candidate invariants that is valid in the initial state and is preserved in the transitions. If the inductive invariant does not entail the target statement, the procedure is repeated with a larger template scheme and larger instances of the system.

The above two approaches need translation of the TM algorithm specification to a transition system. Rewriting a TM algorithm to a transition system is a burden and prone to mistakes. In addition, a rigorous verification needs the proof of equivalence of the TM algorithm specification and its transition system. On the other hand, our program logic can reason on the algorithm specification itself rather than the transition system. As examples of mistakes in the translation of the algorithm specification to transition systems, the following can be noted. There is no visible reads in DSTM algorithm [39] but the specifications of DSTM in [29, 31] and [24] abort the visible readers during the execution of the validate command. As another example, TL2 algorithm [18] is based on version numbers while the specifications of TL2 in [29, 31] and [24] replace the version numbers with the unprecedented notion of modified sets. The definition and proof of equivalence of version numbers to modified sets is missing. Furthermore, there has been a typo of writing os instead of ls in the TL2 transition system in [31]. The follow up work [24] rewrote this specification and incorrectly fixed os to ws and thus verified a different algorithm.

Scalability forced the above two approaches to use abstract models that assume away subtle interleavings of the practical TM algorithms. They modeled blocks of methods as state transitions. Thus, they assumed that fragments of TM methods run atomically and there is no interleaving during the execution of a fragment. The second work [24] assumed further atomicity by unifying two consecutive commands that the first work [31] considered for the commit method of TL2. The presumption that a fragment of a method executes atomically is barely valid in a TM algorithm. In fact, it is the subtle interleavings that render a TM algorithm incorrect. It is also notable that in their specifications of DSTM, aborting visible readers and committing the transaction are done in a single transition and are

thus executed atomically. Therefore, the interleavings that the validation transition should prevent do not happen. Thus, the transition system is correct even without its validation transition. We rewrite the transition systems of [31] and [24] to make them more readable and present them in the appendix section 10.6. While the above two approaches worked on abstract models of TM algorithms, we presented specifications of TM algorithms that are close to their implementation.

Later, Guerraoui et al. [30, 72] considered the fact that fragments of methods cannot be assumed to run atomically. They presented more fine-grained versions of the algorithms in relaxed memory models. But to have the monotonicity property as one of the presumed structural properties, simplified versions of the algorithms were considered and verified. For example, DSTM is specified with no dynamic object allocation while DSTM is fundamentally based on dynamic creation of locator objects. The specification of DSMT does not have any distinction between read and write operations and the read operation simply calls the write operation. This means that similar to writes, reads acquire the location. This is while readers do not acquire the location in DSTM. In their specification, the commit operation writes to every location that is written to during the transaction. This is in contrast to the original algorithm that commits a transaction by a single cas operation. There are simplifications in TL2 as well. The check that the version of the read location is less than the read version is replaced with an equality check. This restricts the concurrency of the algorithm. A local array lver is introduced that is written during the read operations and checked during the commit procedure. This local array does not exist in the original algorithm.

After Guerraoui et al. [31, 30] model-checked abstract version of Intel's McRT STM algorithm, O'Leary et al. [62] applied Spin to model-check a more realistic specification of McRT algorithm. They verified serializability of McRT algorithm for two transactions each consisting of three read or write operations. Baek et al. [4] noted the abstract representation of TM algorithms in the previous works [31, 30]. Particularly, they noticed that the specification of TL2 in [31] does not model the version control mechanism using timestamps. They

emphasized that TM algorithms should be modeled close to the implementation level so that potential bugs are not masked. They presented a model checker to check more realistic specifications of algorithms. The model checker can check TM algorithms that benefit from hardware components or support nested transactions. Using the model checker they could check programs with a small number of transactions and locations. Although they modeled more realistic algorithms, they left relaxed execution for future work. In addition, similar to other works that apply model checking, scalability and state-state explosion is an issue. On the other hand, the semantics of our specification language allows relaxed execution and our program logic can prove the correctness of TM algorithms for arbitrary client programs.

Lesani et al. [50] presented a framework for verification of TM algorithms. Correctness conditions and algorithms are both specified using I/O automata, enabling hierarchical proofs that the algorithms implement the specifications. They used the framework to develop a machine-checked verification of the NORec TM algorithm [16]. The framework is extensible and new proofs can leverage existing ones, eliminating significant work. The framework can be employed to verify realistic algorithms but requires translation of the algorithm to a transition system and more importantly, the process involves manual specification of non-trivial invariants.

Singh [73] developed a runtime verification tool for TM algorithms. He formalizes correctness conditions as a set of conflicts on transactions. The tool builds the conflict graph at runtime and checks that the graph is acyclic. Although it combines coarse and precise runtime analyses to increase the performance of the checker, it leaves the scalability of the tool for future work. In contrast to this work, our approach is full static verification of TM algorithms.

Manovit et al. [53] applied random testing to find bugs in the TCC TM system. They used axiomatic formulation of TM semantics to check the correctness of test runs.

Lourenco et al. [52] ported TL2 from the Sun-pro C compiler, the Solaris operating system and Sparc machines to the gcc compiler, the Linux operating system and Intel x86 32 and

x86 64 architectures. They encountered several bugs during the port and presented traces that exhibit these bugs. They presented test programs that can produce the bug traces. Re-executing the test programs increases the probability that they trigger the bugs.

Both of the above testing works arbitrarily execute the test programs and check if that execution instance exhibits a bug. On the other hand, given a TM algorithm and a bug pattern, our testing approach constructs a trace of the algorithm in the bug pattern if the algorithm can produce a trace in the bug pattern.

8.2 Concurrent Program Logics

Hoare [42] proposed the seminal deductive system to prove the (partial) correctness of sequential imperative programs. Hoare logic presents axioms and deduction rules on triples of pre-condition, statement and post-condition.

Since Hoare [42] proposed the seminal program logic to prove the (partial) correctness of sequential imperative programs, many extensions of it are proposed in attempts to reason about parallel programs.

Owicki and Gries [63] extended Hoare deductive system to reason about parallel programs. Their key idea is that the effect of executing a set of statements in parallel is the same as executing each one by itself, if they do not *interfere*. A statement does not interfere with another statement if and only if every intermediate assertion between atomic actions of the latter is preserved by every atomic action of the former.

The Owicki-Gries definition of interference-freedom is not compositional and therefore the proof technique is not modular. Aiming for a compositional proof technique, Jones [46] modeled interference as a binary relation on states and proposed the *rely/guarantee* deductive system. In this system, each assertion about a statement not only contains a pre-condition and a post-condition but also a *rely* relation modeling the interference from other threads and a *guarantee* relation modeling the interference of this statement for other threads. Nieto

[59] and Coleman and Jones [13] proved soundness of rely/guarantee reasoning. There are strongly related verification methods [57, 37, 2] collectively known as assume/guarantee.

To reason about heap-manipulating programs, Reynolds and others [67, 45] developed *separation logic* as an extension of Hoare logic. Heap-manipulating programs allocate, deallocate, read from and write to heap locations. Separation logic introduces assertions to describe heaps particularly the *separating conjunction* that asserts that the heap can be divided to two separate parts such that each part satisfies its corresponding conjunct. Separation logic supports local reasoning through the *frame rule* that states that the execution of a command does not change if the heap is extended with a separate part.

To enable sharing read-only locations, Boyland [8] introduced fractional *permissions* and later, Bornat and others [7] adapted separation logic to associate permissions with locations. A permission is a value between zero and one. New locations are allocated with the full permission. Writing to but not reading from a location needs its full permission. A location with a definite permission can be split to the separating conjunction of the location with itself such that the permission is divided between the two copies. Dually, permissions are summed when the separating conjunction of the location with itself is merged to a single location.

Separation logic does not allow sharing locations among threads. O'Hearn [60] augmented separation logic with *shared resources* and introduced *concurrent separation logic*. A resource is a set of variables and a resource invariant. Variables of a resource can be accessed only inside conditional critical regions (CCR). CCRs on the same resource execute in mutual exclusion. A CCR can rely on the resource invariant at the entry and should re-establish it at the end. A CCR can move the *ownership* of locations (of the resource invariant) from the resource to the calling thread and vice versa. Brooks [9], Hayman [36] and Vafeiadis [75] defined semantics for and proved the soundness of concurrent separation logic.

Separation logic does not support modular verification of modules and their clients. O'Hearn and others [61] extended separation logic to separately verify the implementations

and the clients of a *module*. The procedures of a module have interface specifications and share a resource invariant. The resource invariant changes if the state representation of the module changes. The *hypothetical frame rule*, a generalization of the classical frame rule, states that if a client of a module is verified using the interface specifications of the module, then it is verified with every resource invariant for the module. Later, Parkinson and Bierman [66] extended the idea with *abstract predicates* to represent interface specifications. Abstract predicates can represent multiple instances of a class and can be extended to reason about class hierarchies.

Concurrent separation logic can reason about locks but it can only model a bounded number of pre-allocated and non-aliased locks and threads. To overcome these limitations, Gotsman and others [27] extended separation logic with *storable locks* and threads. Storable locks can be dynamically allocated, stored in the heap and deallocated. Each lock is allocated as an instance of a *sort*. Each sort is associated with a definite invariant. A thread that acquires a lock gets the lock invariant and should re-establish the invariant when it releases the lock. In other words, the *ownership* of the lock invariant is transfered between the lock itself and accessing threads. In an independent work, Hobor and others [43] extended concurrent separation logic with allocatable locks. A locks is allocated with a definite invariant. Similar to Gotsman and others' work, the ownership invariant is transfered to the thread that locks the lock and is transfered back to the lock when it is unlocked. Later, Buisse and others [10] elaborated the semantics and soundness of these logics.

Rely/guarantee can reason about interfering threads. Separation logic, on the other hand, can separate parts of state and hence supports local reasoning. Vafeiadis and Parkinson [76] aggregated the strengths of the two logics and presented *RGSep* logic. RGSep splits state into shared state that is accessible by all threads and local state which is accessible by a single thread. It uses rely/guarantee to reason about the shared state and separation logic to reason about the local state. The key inference rule is for the critical region (atomic block). The intermediate states of a critical region is not interfered by and does not interfere

other threads. In an independent similar work, Feng and others [26] presented SAGL as an integration of the two logics. Following these two efforts, Feng [25] incorporated separation furthermore to the rely and guarantee conditions and introduced *local rely/guarantee (LRG)* logic. LRG supports local reasoning about separate parts of the shared state. Therefore, it supports modular reasoning not only for the local state but also the shared state. It also allows local sharing of state among a subset of threads.

Rely/guarantee reasoning allows reasoning about the interference of parallel composition of threads. On the other hand, threads are usually dynamically started by *fork* and collected by *join* commands. Dodds and others [21] proposed *deny/guarantee* reasoning that allows interference to be dynamically split to separate parts at the fork command and recombined at the join command. The idea of separation of interference is inspired by separation of state in separation logic. Interference is described using deny and guarantee permissions: a deny permission specifies that the environment does not do an action and a guarantee permission specifies that the current thread can do an action.

A data structure can be represented with a single abstract predicate [66]. However, multiple fine-grained abstract predicates are often needed to refer to the same data structure. Dinsdale-Young and others [20] presented *concurrent abstract predicates* that allow multiple abstractions in the presence of sharing. The definition of a concurrent abstract predicate specifies both the permitted actions and the state of the shared data structure. The definition of the predicates are used to verify the implementation of the data structure. The predicates are used to separately verify clients.

Chapter 9

Conclusions and Future Works

We introduced an architecture-independent specification language for synchronization algorithms. We presented the specification of several TM algorithms. We hope that other researchers represent algorithms in the language. The language can serve as a shared platform for development of verification benchmark suites of synchronization algorithms. These suites can facilitate comparison of verification techniques. Compilers can be developed that optimize the translation of algorithm specifications to particular operating systems and architectures. Particularly, fence allocation is an interesting future research direction.

We introduced the markability correctness condition as the conjunction of intuitive invariants: write-observation and read-preservation. We proved the equivalence of markability and opacity correctness conditions. A future research direction is to study the inherent difficulty of each of the invariants and the interplay between the invariants and the liveness properties of transactional memory.

We have identified two bug patterns that lead to non-opacity. Samand is flexible and can accommodate a variety of bug patterns such as H_{WE2} that was suggested by a DISC reviewer. Samand outputs an execution trace of McRT that matches H_{WE2} in about 7 minutes. Our tool handles small bug patterns efficiently; scalability is left for future work. We hope that our observations and tool can help TM algorithm designers to avoid the write-skew, write-

exposure, and other pitfalls. We envision a methodology in which TM algorithm designers use Samand during the design to avoid known pitfalls. Samand can be used also during maintenance of TM algorithms. For example, a set of bug patterns can serve as a regression test suite. Additionally, our tool can be used to avoid pitfalls in other synchronization algorithms.

We presented synchronization object logic (SOL) that supports reasoning about the execution order and linearization orders of method calls. We proved the soundness of the logic. Future research can study the completeness of the logic. We proved that the derivation of markability for an algorithm specification is a syntactic proof of its opacity. We used SOL to prove the markability of the TL2 algorithm in PVS. Future work can prove the markability of other TM algorithms. Furthermore, the applicability of the logic to prove the linearizability of concurrent data structures can be studied. Variants of the logic can be applied to prove liveness properties as well.

Chapter 10

Appendix

10.1 Synchronization Object Language

10.1.1 Specification

Let us define data and control dependency of statements. We define the context $\mathcal{R}$ as follows. [ ] denotes a hole.

$$\begin{array}{rcll} \mathcal{R} & ::= & \textbf{if } b\ \mathcal{R}\ \textbf{else } s \mid \textbf{if } b\ s\ \textbf{else } \mathcal{R} \mid & \text{Context} \\ & & \mathcal{R},\ s \mid s,\ \mathcal{R} \mid & \\ & & [\,] & \end{array}$$

A statement is data dependent to a method call if it accesses the return value of the method call. In a statement s, the statement s' is data-dependent on the method call labeled c if there exists x, o, n, τ, u, $\mathcal{R}_1$, and $\mathcal{R}_2$, such that

$s = \mathcal{R}_1[c_1 \triangleright\ x = o.n_\tau(u)], \mathcal{R}_2[s']$

and x appears in s'.

Every statement in the scope of an if statement is control-dependent on the if statement. The statement s' is control-dependent on the if statement s if there exists b, $\mathcal{R}$ and s'' such

that

$s = \textbf{if } b\ \mathcal{R}[s']\ \textbf{else } s''$ or

$s = \textbf{if } b\ s''\ \textbf{else } \mathcal{R}[s']$.

Every statement before a return statement in the program is control-dependent to it. In a statement s, the statement s' is control-dependent on the return statement labeled c if there exists u, $\mathcal{R}_1$, and $\mathcal{R}_2$, such that

$s = \mathcal{R}_1[s'], \mathcal{R}_2[c \triangleright\ \textbf{return } u]$

Let $Calls_\pi(\phi, n)$ denote the set of labels of call statements in π where the method name n is called on the base object name ϕ.

$$Calls_\pi(\phi, n) = \{c \mid c \in Labels(\pi)\ \wedge\ basename(obj_\pi(c)) = \phi\ \wedge\ name_\pi(c) = n\} \quad (10.1)$$

Consider a method definition n of a specification π.

$$\textbf{def } n_t(x^*)\ s, r$$

The set $PreReturns_\pi(c)$ is the set of labels of the return statements before c in the body of n.

$$\begin{aligned} PreReturns_\pi(c) = \{c' \mid \exists \mathcal{R}_1, \mathcal{R}_2, u, q, x, o, n, \tau, u' : \\ s = \mathcal{R}_1[c' \triangleright\ \textbf{return } u], \mathcal{R}_2[q] \wedge \\ q = (c \triangleright\ x = o.n_\tau(u'))\ \vee\ q = (c \triangleright\ \textbf{return } u')\} \end{aligned} \quad (10.2)$$

10.1.2 Semantics

10.1.2.1 Execution History

Lemma 1:

We Assume

(1) $l \prec_X l'$

From [1] and definition of $\sim_X$, we have

(2) $\neg(l' \sim_X l)$

From [1], we have

(3) $rEv(l) \lhd_X iEv(l')$

As X is a valid history, we have

(4) $iEv(l) \lhd_X rEv(l)$

(5) $iEv(l') \lhd_X rEv(l')$

From [4], [3], and [5], we have

(6) $iEv(l) \lhd_X rEv(l')$

From [6], we have

(7) $\neg(rEv(l') \lhd_X iEv(l))$

From [7], and definition of $\prec_X$, we have

(9) $\neg(l' \prec_X l)$

From [3] and [7], we have

(9) $\neg(l' = l)$

Lemma 2:

Straightforward from the definition of $\prec_X$.

Lemma 3:

We have

(1) $l_1 \prec_X l_2$

(2) $l_3 \prec_X l_4$

(3) $l_2 \sim_X l_3$

From [1], we have

(4) $rEv(l_1) \lhd_X iEv(l_2)$

From [2], we have

(5) $rEv(l_3) \lhd_X iEv(l_4)$

From [3], we have

(6) $\neg(l_3 \prec_X l_2)$

From [6], we have

(7) $\neg(rEv(l_3) \lhd_X iEv(l_2))$

From [7], we have

(8) $iEv(l_2) \lhd_X rEv(l_3)$

From [4], [8], and [5], we have

(9) $rEv(l_1) \lhd_X iEv(l_4)$

From [9], we have

$l_1 \prec_X l_4$

Lemma 4:

Straightforward from the definition of $\prec_X$ and $\sim_X$.

Lemma 5:

Straightforward from the definition of $\prec_X$.

Lemma 6:

Straightforward from the definition of $\prec_X$ and $\lhd_X$.

Lemma 7:

Straightforward from the definition of $\prec_X$ and $\lhd_X$.

10.1.2.2 Synchronization Object Types

Lemma 8:

Straightforward from $\prec_X \subseteq \prec_L$.

Lemma 9:

Straightforward from Lemmas 13, [4], 8, and 10.

Lemma 10:

We have

(1) $l \prec_L l'$

From [1], we have

(2) $rEv(l) \lhd_L iEv(l')$

From the well-formedness of the history O,

we have

(3) $iEv(l) \lhd_L rEv(l)$

(4) $iEv(l') \lhd_L rEv(l')$

From [3], [2] and [4], we have

(5) $iEv(l) \lhd_L rEv(l')$

From [5], we have

(6) $\neg(rEv(l') \lhd_L iEv(l))$

From [2] and [6], we have

(7) $\neg(l' = l)$

From the definition of $\prec_X$ on [6], we have

(8) $\neg(l' \prec_L l)$

The conclusion is

[8] and [7]

Lemma 11:

Straightforward from the fact that L is a member of sequential specification and a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.

Lemma 12:

Straightforward from the fact that L is a member of sequential specification and a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.

We have

(1) $l \in X$

(2) $l' \in X$

(3) $X \equiv L$

(4) $L \in SeqSpec(o)$

From [4], we have

(5) $L \in Sequential$

From [3], [1] and [2], we have

(6) $l \in L$

(7) $l' \in L$

From [4], [6] and [7], we have

$l \prec_L l' \ \vee \ l' \prec_L l \ \vee \ l = l'$

Lemma 13:

Straightforward from the fact that L is equivalent to X.

We have

(1) $X \equiv L$

(2) $L \in SeqSpec(o)$

(3) $l \prec_L l'$

From [3], we have

(4) $l \in L$

(5) $l' \in L$

From [2] on [4] and [5], we have

(6) $obj_L(l) = o$

(7) $obj_L(l') = o$

From [1] on [4] and [5], we have

$l \in X$

$l' \in X$

From [1] on [6] and [7], we have

$obj_X(l) = o$

$obj_X(l') = o$

Lemma 14:

Using L2X and XTotal, we have four cases:

Case: $l \prec l'$

Straightforward from XTrans.

Case: $l \sim l'$

Straightforward from XXTrans.

Case: $l' \prec l$

Straightforward from X2L and LASym.

Case: $l' = l$

Straightforward from LASym.

Lemma 15:

Derived from the semantics of basic objects (Definition 1) and the sequential specification of register (Definition 4).

Lemma 16:

Derived from the semantics of basic register (Definition 5).

Lemma 17:

This is a restatement of Theorem 3 from the original definition of linearizability []. Derivable from the semantics of linearizable objects (Definition 3) and the sequential specification of register (Definition 4).

Lemma 18:

Derivable from the semantics of linearizable objects (Definition 3) and the sequential specification of cas register (Definition 6).

Lemma 19:

Derivable from the semantics of linearizable objects (Definition 3) and the sequential specification of cas register (Definition 6).

Lemma 20:

Derivable from the semantics of linearizable objects (Definition 3), the sequential specification of the lock (Definition 7), the owner-respecting property (Definition 8), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 21:

Derived from Lemma 20.

Lemma 22:

Derived from Lemma 20 and the sequential specification of lock (Definition 7).

Lemma 23:

Derived from Lemma 20 and the sequential specification of lock (Definition 7).

Lemma 24:

Derived from Lemma 20 and the sequential specification of lock (Definition 7).

Lemma 25:

Derivable from the semantics of linearizable objects (Definition 3), the sequential specification of the lock (Definition 9), the owner-respecting property (Definition 25), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 26:

Derived from Lemma 25.

Lemma 27:

Derived from Lemma 25 and the sequential specification of try-lock (Definition 9).

Lemma 28:

Derived from Lemma 25 and the sequential specification of try-lock (Definition 9).

Lemma 29:

Derived from Lemma 25 and the sequential specification of try-lock (Definition 9).

Lemma 30:

Derivable from the semantics of linearizable objects (Definition 3), the sequential specification of counter (Definition 11).

Lemma 31:

Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 12).

Lemma 32:

Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 12).

Lemma 33:

Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 13).

Lemma 34:

Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 13).

10.2 TM Correctness

10.2.1 The Marking Theorem

For the sake of brevity, we use the shorthand notation

$\exists l = o.n_T(v_1){:}v_2 \in X$

for

$\exists l \in X : obj_X(l) = o \ \wedge \ name_X(l) = n \ \wedge \ thread_X(l) = T \ \wedge \ arg1_X(l) = v_1 \ \wedge \ retv_X(l) = v_2$

and similarly for universal quantification.

We also use W, R to denote labels.

Lemma 42. *For all $S \in TSequential$, $T \in S$, $S' = Visible(S, T)$, and $T', T'' \in S'$, we have $T' \preceq_{S'} T'' \Leftrightarrow T' \preceq_S T''$.*

Proof.

$$
\begin{aligned}
& T' \preceq_{S'} T'' \\
\Leftrightarrow \ & S'|T' \lhd_{S'} S'|T'' \ \vee \ T' = T'' \\
\Leftrightarrow \ & S|T' \lhd_{S'} S|T'' \ \vee \ T' = T'' \\
\Leftrightarrow \ & S|T' \lhd_S S|T'' \ \vee \ T' = T'' \\
\Leftrightarrow \ & T' \preceq_S T''
\end{aligned}
$$

In these four steps we apply:

1) the definition of $\preceq_{S'}$,

2) that the definition of $Visible(S, T)$ implies both $S'|T' = S|T'$ and $S'|T'' = S|T''$,

3) $S' \Subset S$, and

4) the definition of $\preceq_S$. □

Lemma 43. *For all* $S \in TSequential$, $T \in S$, $i \in I$, $v, v' \in V$, $R = read_T(i){:}v \in GlobalTReads(S)$, $S' = Visible(S,T)$, $T' \in S'$, *and* $W' = write_{T'}(i,v') \in GlobalTWrites(S)$, *we have*

$$NoWriteBetween_{(S'|i)}(W', R) \quad \Leftrightarrow \quad NoWriterBetween_{S,i}(T', \preceq\!\!\!\!\!\!\ll_S, T)$$

Proof.

$$\begin{aligned}
& NoWriteBetween_{(S'|i)}(W', R) \\
\Leftrightarrow\ & \forall W'' \in TWrites(S'|i)\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\Leftrightarrow\ & \forall T'' \in S'|i\colon \forall i' \in I\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i', v'') \in S'|i\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\Leftrightarrow\ & \forall T'' \in S'|i\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'|i\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\Leftrightarrow\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon W'' \preceq_{S'} W' \vee R \preceq_{S'} W'' \\
\Leftrightarrow\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \preceq\!\!\!\!\!\!\ll_{S'} T' \vee T \preceq\!\!\!\!\!\!\ll_{S'} T'' \\
\Leftrightarrow\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \preceq\!\!\!\!\!\!\ll_{S} T' \vee T \preceq\!\!\!\!\!\!\ll_{S} T'' \\
\Leftrightarrow\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \lessdot_S T \Rightarrow T'' \preceq\!\!\!\!\!\!\ll_S T' \\
\Leftrightarrow\ & \forall T'' \in S\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S\colon \\
& \qquad [[(T'' = T) \vee (T'' \lessdot_S T \wedge T'' \in Committed(S))] \wedge [T'' \lessdot_S T]] \Rightarrow T'' \preceq\!\!\!\!\!\!\ll_S T' \\
\Leftrightarrow\ & \forall T'' \in S\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S\colon \\
& \qquad (T'' \in Committed(S) \wedge\ T'' \lessdot_S T) \Rightarrow T'' \preceq\!\!\!\!\!\!\ll_S T' \\
\Leftrightarrow\ & \forall T'' \in Writers_S(i)\colon T'' \lessdot_S T \Rightarrow T'' \preceq\!\!\!\!\!\!\ll_S T' \\
\Leftrightarrow\ & \forall T'' \in Writers_S(i)\colon T'' \preceq\!\!\!\!\!\!\ll_S T' \vee T \preceq\!\!\!\!\!\!\ll_S T'' \\
\Leftrightarrow\ & NoWriterBetween_{S,i}(T', \preceq\!\!\!\!\!\!\ll_S, T)
\end{aligned}$$

In these twelve steps, we apply:

1) the definition of $NoWriteBetween$,

2) the definition of $Writes$,

3) the definition of projection $S'|i$,

4) R, W' and W'' access location i,

5) $S' \in TSequential$ and $R \in GlobalTReads(S')$ and $W' \in GlobalTWrites(S')$ (that are concluded from $S \in TSequential$, $R \in GlobalTReads(S)$, $W' \in GlobalTWrites(S)$ and $S' = Visible(S, T)$.),

6) Lemma 42,

7) Boolean logic and that $\preceq_S$ is total,

8) the definition of $Visible$,

9) logical simplification,

10) the definition of $Writers$,

11) Boolean logic and that $\preceq_S$ is total, and

12) the definition of $NoWriterBetween$. □

Lemma 44. $TSequential \subset Sequential$

Proof. Straightforward from definitions of $TSequential$, $THistory$ and $Sequential$. □

Lemma 45. $\forall i \in I\colon\ \forall v, v' \in V\colon\ \forall T, T' \in Trans\colon$ *if* $R = read_T(i){:}v$, $W = write_{T'}(i, v)$, $W' = write_T(i, v')$, $S \in TSequential$, $W \prec_S R$, $NoWriteBetween_S(W, R)$ *and* $W' \prec_S R$, *then* $T = T'$.

Proof. Suppose (1) $S \in TSequential$, (2) $W \prec_S R$, (3) $NoWriteBetween_S(W, R)$ and (4) $W' \prec_S R$. From [1] and Lemma 44, we have (5) $S \in Sequential$. From [4] and [5], we have (6) $\neg(R \prec_S W')$. From [3] we have (7) $W' \preceq_S W \vee R \prec_S W'$. From [6] and [7], we have (8) $W' \preceq_S W$. From [2] and [8], we have (9) $W' \preceq_S W \preceq_S R$. From [9], [1], and that W' and R are by T and W is by T', we have $T = T'$. □

Lemma 46. *Suppose $S \in TSequential$. We have:*

$$\begin{aligned}
&\forall T \in S : \forall i \in I : \forall v \in V : \forall R = read_T(i) : v \in LocalTReads(S) : \\
&\qquad \exists T' \in Visible(S,T) : \exists W = write_{T'}(i,v) \in Visible(S,T) : \\
&\qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge \ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\Leftrightarrow \quad & LocalWriteObs(S)
\end{aligned}$$

Proof. Suppose $S \in TSequential$. Thus, from Lemma 44, we have $S \in Sequential$. Let $S' = Visible(S,T)$. From $S \in TSequential$ and Lemma 42, we have $S' \in TSequential$. Thus, from Lemma 44, we have $S' \in Sequential$. From the definition of $Visible$, we have

$S'|T = S|T$.

$$
\begin{aligned}
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists T' \in S'\colon \exists W = write_{T'}(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists v' \in V\colon \exists W' = write_T(i,v') \in S\colon W' \prec_S R \ \wedge \\
& \qquad \exists T' \in S'\colon \exists W = write_{T'}(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists v' \in V\colon \exists W' = write_T(i,v') \in S'\colon W' \prec_S R \ \wedge \\
& \qquad \exists T' \in S'\colon \exists W = write_{T'}(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists v' \in V\colon \exists W' = write_T(i,v') \in S'\colon W' \prec_{S'} R \ \wedge \\
& \qquad \exists T' \in S'\colon \exists W = write_{T'}(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists v' \in V\colon \exists W' = write_T(i,v') \in S'\colon W' \prec_{(S' \mid i)} R \ \wedge \\
& \qquad \exists T' \in S'\colon \exists W = write_{T'}(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists v' \in V\colon \exists W' = write_T(i,v') \in S'\colon W' \prec_{(S' \mid i)} R \ \wedge \\
& \qquad \exists W = write_T(i,v) \in S'\colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \ \wedge\ NoWriteBetween_{(S' \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists W = write_T(i,v) \in S'\colon \\
& \qquad\qquad \ldots
\end{aligned}
$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S'} R \ \wedge \ NoWriteBetween_{(S' \mid i)}(W,R)$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ NoWriteBetween_{(S' \mid i)}(W,R)$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \forall W' \in TWrites(S' \mid i)\colon W' \preceq_{(S' \mid i)} W \vee R \prec_{(S' \mid i)} W'$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \neg\exists W' \in TWrites(S' \mid i)\colon \neg(W' \preceq_{(S' \mid i)} W) \ \wedge \ \neg(R \prec_{(S' \mid i)} W')$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \neg\exists W' \in TWrites(S' \mid i)\colon W \prec_{(S' \mid i)} W' \prec_{(S' \mid i)} R$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \neg\exists v' \in V\colon \exists W' = write_T(i,v')\colon W \prec_{(S' \mid i)} W' \prec_{(S' \mid i)} R$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \neg\exists v' \in V : \exists W' = write_T(i,v')\colon W \prec_{(S \mid i)} W' \prec_{(S \mid i)} R$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \neg\exists W' \in TWrites(S \mid i)\colon W \prec_{(S \mid i)} W' \prec_{(S \mid i)} R$$

$$\Leftrightarrow \quad \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in LocalTReads(S)\colon$$

$$\exists W = write_T(i,v) \in S\colon$$

$$W \prec_{S} R \ \wedge \ \forall W' \in TWrites(S \mid i)\colon \neg(W \prec_{(S \mid i)} W') \ \vee \ \neg(W' \prec_{(S \mid i)} R)$$

In these twenty steps, we apply: 1) the definition of $LocalReads$,

2) the definition of $Visible$,

3) $S'|T = S|T$ and that both W' and R are by T,

4) that both W' and R are on i,

5) Lemma 45,

6) duplicate conjunction,

7) the definition of $Visible$,

8) that both R and W are on i,

9) $S'|T = S|T$ and that both R and W are by T,

10) the definition of $NoWriteBetween$,

11) first-order logic,

12) $(S' \mid i) \in Sequential$,

13) from $(S' \mid i) \in TSequential$, R and W are by transaction T and W' is between them, we have W' is by T,

14) $S'|T = S|T$,

15) from $(S \mid i) \in TSequential$, R and W are by transaction T and W' is between them, we have W' is by T.

16) first-order logic,

17) $(S \mid i) \in Sequential$,

18) $(S \mid i) \in Sequential$, $thread_H(R) = thread_H(W) = T$ and $arg1_H(R) = arg1_H(W) = i$,

19) the definition of $NoWriteBetween$,

20) the definition of $LocalWriteObs$.

□

Lemma 47. *Suppose* $S \in TSequential \cap TComplete$. *We have:*

$$
\begin{aligned}
& S \in TSeqSpec \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S : \forall i \in I : \forall v \in V : \forall R = read_T(i){:}v \in GlobalTReads(S) : \\
& \exists T' \in Committed(S) : \exists W = write_{T'}(i, v) \in GlobalTWrites(S) : \\
& \quad (T' \lessdot_S T)\ \wedge\ NoWriterBetween_{S,i}(T', \preceq_S, T)
\end{aligned}
$$

Proof. Suppose $S \in TSequential \cap TComplete$. From $S \in TSequential$ and Lemma 42, we

have $Visible(S,T) \in TSequential$.

$$
\begin{aligned}
& S \in TSeqSpec \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon (Visible(S,T) \mid i) \in SeqSpec(i) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \\
& \qquad \forall T'' \in (Visible(S,T) \mid i)\colon \forall v \in V\colon \forall R = read_{T''}(i){:}v \in (Visible(S,T) \mid i)\colon \\
& \qquad \exists T' \in (Visible(S,T) \mid i)\colon \exists W = write_{T'}(i,v) \in (Visible(S,T) \mid i)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \\
& \qquad \forall T'' \in Visible(S,T)\colon \forall v \in V\colon \forall R = read_{T''}(i){:}v \in Visible(S,T)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_{T}(i){:}v \in S\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\Leftrightarrow\ & \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_{T}(i){:}v \in LocalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
& \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_{T}(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\Leftrightarrow\ & LocalWriteObs(S) \ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_{T}(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \ \wedge\ NoWriteBetween_{(Visible(S,T) \mid i)}(W,R)
\end{aligned}
$$

$$
\begin{aligned}
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad W \prec_{Visible(S,T)} R\ \wedge\ NoWriteBetween_{(Visible(S,T)\ |\ i)}(W,R) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad T' \lessdot_{Visible(S,T)} T\ \wedge\ NoWriteBetween_{(Visible(S,T)\ |\ i)}(W,R) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in Visible(S,T)\colon \\
& \qquad T' \lessdot_S T\ \wedge\ NoWriteBetween_{(Visible(S,T)\ |\ i)}(W,R) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \qquad T' \lessdot_S T\ \wedge\ NoWriteBetween_{(Visible(S,T)\ |\ i)}(W,R) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \qquad T' \lessdot_S T\ \wedge\ NoWriterBetween_{S,i}(T', \preceq_S, T) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Visible(S,T)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \qquad (T' \lessdot_S T)\ \wedge\ T' \in Committed(S)\ \wedge\ NoWriterBetween_{S,i}(T', \preceq_S, T) \\
\Leftrightarrow\ & LocalWriteObs(S)\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \qquad \exists T' \in Committed(S)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon
\end{aligned}
$$

In these thirteen steps, we apply:

1) the definition of $TSeqSpec$ and $S \in TSequential \cap TComplete$,

2) the definition of $SeqSpec(i)$,

3) R and W access location i,

4) that we can choose $T'' = T$,

5) $TReads(S) = LocalTReads(S) \cup GlobalTReads(S)$,

6) Lemma 46,

7) that R and W are both on location i

8) that R and W are by transactions T and T' respectively, $Visible(S,T) \in TSequential$, and $R \in GlobalTReads(Visible(S,T))$ (because $R \in GlobalTReads(R)$ and $Visible(S,T)|T = S|T$),

9) Lemma 42,

10) $T' \lessdot_S T$ and $NoWriteBetween_{(Visible(S,T)\ |\ i)}(W,R)$,

11) Lemma 43,

12) $T' \in Visible(S,T)$ and $(T' \lessdot_S T)$, and

13) the definition of $Visible(S,T)$. □

Lemma 48. (Invariance) *If $H \equiv H'$, then for every marking relation $\sqsubseteq$, $Marking(H) = Marking(H')$ and $ReadPres(H, \sqsubseteq) \Leftrightarrow ReadPres(H', \sqsubseteq)$ and $WriteObs(H, \sqsubseteq) \Leftrightarrow WriteObs(H', \sqsubseteq$).*

Proof. Immediate from the definitions of $Marking$, $ReadPres$, and $WriteObs$. □

Lemma 49. $\forall H \in THistory\colon \forall \sqsubseteq \in Marking(H)\colon \exists S \in TSequential\colon H \equiv S \ \wedge \ \preccurlyeq_H \subseteq \preccurlyeq_S \ \wedge \ \preccurlyeq_S \subseteq \sqsubseteq$.

Proof. Let $H \in THistory$ and let $\sqsubseteq \in Marking(H)$. We have that $\sqsubseteq$ is a total order of $Trans$ so we can choose a permutation π on $1..n$ such that $\forall i, j \in 1..n\colon (i < j) \Leftrightarrow (T_{\pi(i)} \sqsubset T_{\pi(j)})$. Define: $S = H|T_{\pi(1)}, \ldots, H|T_{\pi(n)}$. It is straightforward to prove that $S \in TSequential \ \wedge \ H \equiv S \ \wedge \ \preccurlyeq_H \subseteq \preccurlyeq_S \ \wedge \ \preccurlyeq_S \subseteq \sqsubseteq$. □

Lemma 50. *Suppose $\sqsubseteq \in Marking(H) \ \wedge \ p_2 \notin Writers_H(i)$.*
If $NoWriterBetween_{H,i}(T_1, \sqsubseteq, p_2)$ and $NoWriterBetween_{H,i}(p_2, \sqsubseteq, T_3)$,
then $NoWriterBetween_{H,i}(T_1, \sqsubseteq, T_3)$.

Proof.

$$
\begin{aligned}
& NoWriterBetween_{H,i}(T_1, \sqsubseteq, p_2) \ \wedge \ NoWriterBetween_{H,i}(p_2, \sqsubseteq, T_3) \\
\Leftrightarrow \quad & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1 \ \vee \ p_2 \sqsubseteq T) \ \wedge \ (T \sqsubseteq p_2 \ \vee \ T_3 \sqsubseteq T) \\
\Leftrightarrow \quad & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1 \ \wedge \ (T \sqsubseteq p_2 \ \vee \ T_3 \sqsubseteq T)) \ \vee \\
& \qquad\qquad (p_2 \sqsubseteq T \ \wedge \ T \sqsubseteq p_2) \ \vee \ (p_2 \sqsubseteq T \ \wedge \ T_3 \sqsubseteq T) \\
\Longrightarrow \quad & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1) \ \vee \ (T_3 \sqsubseteq T) \\
\Leftrightarrow \quad & NoWriterBetween_{H,i}(T_1, \sqsubseteq, T_3)
\end{aligned}
$$

The first step uses the definition of $NoWriterBetween$. The second step uses $\wedge$ distribution over $\vee$. The third step simplifies the first disjunct using conjunction elimination, eliminates the second disjunct using $p_2 \notin Writers_H(i)$ and simplifies the third disjunct using conjunction elimination. The fourth step uses the definition of $NoWriterBetween$. □

Lemma 51. *Suppose $S \in TSequential \cap TComplete$. We have:*

$$S \in TSeqSpec \Leftrightarrow S \in FinalStateMarkable$$

Proof. Let $S \in TSequential \cap TComplete$. From Lemma 47, the definition of $FinalStateMarkable$, and $S \in TComplete$, we have that we must prove:

$$\begin{aligned}
& LocalWriteObs(S) \ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \exists T' \in Committed(S)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \quad (T' \prec\!\!\!\prec_S T) \ \wedge \ NoWriterBetween_{S,i}(T', \preceq\!\!\!\!\preceq_S, T) \\
\Leftrightarrow \ & \exists \sqsubseteq \in Marking(S)\colon \ \ \preceq\!\!\!\!\preceq_S \subseteq \sqsubseteq \ \ \wedge \ \sqsubseteq \in ReadPres(S) \wedge \ \sqsubseteq \in WriteObs(S)
\end{aligned}$$

From the definition of $WriteObs$, $GlobalWriteObs$ and $LastPreAccessor$ we have that:

$$\begin{aligned}
& WriteObs(S, \sqsubseteq) \\
\Leftrightarrow \ & LocalWriteObs(S) \ \wedge \\
& \forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \exists T' \in Trans\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \quad T' \in Writers_S(i) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R) \\
\Leftrightarrow \ & LocalWriteObs(S) \ \wedge \\
& \forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
& \exists T' \in Trans\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
& \quad T' \in Committed(S) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R)
\end{aligned}$$

We are now ready to prove the two directions of the equivalence.

$\Rightarrow$:

Assume that

$$
\begin{array}{l}
LocalWriteObs(S)\ \wedge \\
\forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
\exists T' \in Committed(S)\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
\quad (T' \prec\!\!\!\prec_S T)\ \wedge\ NoWriterBetween_{S,i}(T', \preceq\!\!\!\!\!\!\prec_S, T)
\end{array}
$$

Define:

$$
\begin{array}{rcl}
p_1 \sqsubset p_2 & \Leftrightarrow & (p_1 \prec\!\!\!\prec_S p_2)\ \vee \\
 & & (thread_S(p_1) \preceq\!\!\!\!\!\!\prec_S p_2)\ \vee \\
 & & (p_1 \preceq\!\!\!\!\!\!\prec_S thread_S(p_2)) \\
p_1 \sqsubseteq p_2 & \Leftrightarrow & p_1 \sqsubset\ \vee\, p_2 p_1 = p_2
\end{array}
$$

We show that

$$
\begin{array}{l}
\sqsubseteq \in Marking(S)\ \wedge \\
\preceq\!\!\!\!\!\!\prec_S \subseteq \sqsubseteq\ \ \wedge\ \sqsubseteq \in ReadPres(S)\ \wedge \\
LocalWriteObs(S)\ \wedge \\
\forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalTReads(S)\colon \\
\exists T' \in Trans\colon \exists W = write_{T'}(i,v) \in GlobalTWrites(S)\colon \\
\quad T' \in Committed(S)\ \wedge\ T' \neq T\ \wedge\ T' \sqsubset R\ \wedge\ NoWriterBetween_{S,i}(T', \sqsubseteq, R)
\end{array}
$$

It is straightforward to prove $\sqsubseteq \in Marking(S)$ and $\preceq\!\!\!\!\!\!\prec_S \subseteq \sqsubseteq$, $ReadPres(S, \sqsubseteq)$. Additionally, the first conjunct of $WriteObs(S, \sqsubseteq)$ (that is, $LocalWriteObs(S)$) is immediate from the assumption. So, we still need to prove the second conjunct of $WriteObs(S, \sqsubseteq)$.

Let $T \in Trans$, $i \in I$, $v \in V$, $R = read_T(i){:}v \in GlobalTReads(S)$. From the assump-

tion (the left-hand side), we have that we can find (1) $T' \in Committed(S)$ and (2) $W = write_{T'}(i, v) \in GlobalTWrites(S)$ such that (3) $(T' \lessdot_S T)$ and (4) $NoWriterBetween_{S,i}(T', \preceq \preceq_S, T)$. Let us now prove each conjunct of $T' \neq T \ \wedge \ T' \sqsubseteq R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq , R)$ in turn.

From [3] and that $\preceq_S$ is a total order of $Trans(S)$, we have (5) $T' \neq T$. From [3] and the definition of $\sqsubseteq$, we have $T' \sqsubseteq R$. From [4] and $\lessdot_S \subseteq \sqsubseteq$, we have (6) $NoWriterBetween_{S,i}(T', \sqsubseteq , T)$. From $T \lessdot_S T$ and the definition of $\sqsubseteq$, we have (7) $R \sqsubseteq T$. From [6], [7] and the definition of $\sqsubseteq$ and transitivity of $\lessdot_S$, we have $NoWriterBetween_{S,i}(T', \sqsubseteq, R)$.

$\Leftarrow$:

Assume the right-hand side and choose $\sqsubseteq \in Marking(S)$ such that:

$$
\begin{array}{l}
\lessdot_S \subseteq \sqsubseteq \ \ \wedge \ \sqsubseteq \in ReadPres(S) \ \wedge \\
S \in TLocalSeqSpec \ \wedge \\
\forall T \in Trans \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in GlobalTReads(S) \colon \\
\exists T' \in Committed(S) \colon \exists \, W = write_{T'}(i, v) \in GlobalTWrites(S) \colon \\
\quad T' \neq T \ \wedge \ T' \sqsubseteq R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R)
\end{array}
$$

We show that

$$
\begin{array}{l}
LocalWriteObs(S) \ \wedge \\
\forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in GlobalTReads(S) \colon \\
\exists T' \in Committed(S) \colon \exists W = write_{T'}(i, v) \in GlobalTWrites(S) \colon \\
\quad (T' \lessdot_S T) \ \wedge \ NoWriterBetween_{S,i}(T', \lessdot_S, T)
\end{array}
$$

The first conjunct (of the left-hand side), $LocalWriteObs(S)$, is immediate from the assumption. From the assumption we have (1) $\lessdot_S \subseteq \sqsubseteq$, (2) $\sqsubseteq \in ReadPres(S)$. Let $T \in Trans$, $i \in I$, $v \in V$, $R = read_T(i){:}v \in GlobalTReads(S)$. From the above property of $\sqsubseteq$, we have

that we can find (3) $T' \in Committed(S)$ and (4) $W = write_{T'}(i, v) \in GlobalTWrites(S)$ such that (5) $T' \neq T$ and (6) $T' \sqsubseteq R$ and (7) $NoWriterBetween_{S,i}(T', \sqsubseteq, R)$. From [1], that $\sqsubseteq$ is a total order on $Trans(S)$ ($\sqsubseteq \in Marking(S)$), and that $\preceq_S$ is a total order on $Trans(S)$ ($S \in TSequential$), we have (8) $\forall T, T' \in Trans\colon T' \sqsubseteq T \Rightarrow T' \preceq_S T$.

First we prove $T' \prec_S T$. From [2] ,we have (9) $NoWriterBetween_{S,i}(T, \sqsubseteq, R)$. From [3] and [4], we have (10) $T' \in Writers_S(i)$. From [9] and [10], we have (11) $T' \sqsubseteq T \ \vee \ R \sqsubseteq T'$. From [6], $T' \neq R$ and $\sqsubseteq$ is a total order on $\{R\} \cup Writers_S(i)$ ($\sqsubseteq \in Marking(S)$), we have (12) $R \not\sqsubseteq T'$. From [11] and [12], we have (13) $T' \sqsubseteq T$. From [8] and [13], we have (14) $T' \preceq_S T$. From [14] and [5], we have $T' \prec_S T$.

Second, we prove $NoWriterBetween_{S,i}(T', \preceq_S, T)$. From [2], we have (15) $NoWriterBetween_{S,i}(R, \sqsubseteq , T)$. From $R \notin Writers_S(i)$, [7], [15], and Lemma 50, we have (16) $NoWriterBetween_{S,i}(T', \sqsubseteq , T)$. From [16] and [8] we have $NoWriterBetween_{S,i}(T', \preceq_S, T)$. □

Theorem (Marking) $FinalStateOpaque = FinalStateMarkable$.

Proof.

$$
\begin{aligned}
& FinalStateOpaque \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists S \in TSequential \colon \\
& \qquad H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge S \in TSeqSpec\} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists S \in TSequential \colon \\
& \qquad H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge S \in FinalStateMarkable\} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists S \in TSequential \colon H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge \\
& \quad \exists \sqsubseteq \in Marking(S) \colon \; \preccurlyeq_S \subseteq \sqsubseteq \; \wedge ReadPres(S, \sqsubseteq) \; \wedge \; WriteObs(S, \sqsubseteq)\} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists S \in TSequential \colon H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge \\
& \quad \exists \sqsubseteq \in Marking(H') \colon \; \preccurlyeq_S \subseteq \sqsubseteq \; \wedge ReadPres(H', \sqsubseteq) \; \wedge \; WriteObs(H', \sqsubseteq)\} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists \sqsubseteq \in Marking(H') \colon \\
& \quad ReadPres(H', \sqsubseteq) \; \wedge \; WriteObs(H', \sqsubseteq) \; \wedge \\
& \quad \exists S \in TSequential \colon H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge \; \preccurlyeq_S \subseteq \sqsubseteq \; \} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists \sqsubseteq \in Marking(H') \colon \\
& \quad \preccurlyeq_{H'} \subseteq \sqsubseteq \; \wedge ReadPres(H', \sqsubseteq) \; \wedge \; WriteObs(H', \sqsubseteq) \; \wedge \\
& \quad \exists S \in TSequential \colon H' \equiv S \wedge \; \preccurlyeq_{H'} \subseteq \preccurlyeq_S \; \wedge \; \preccurlyeq_S \subseteq \sqsubseteq \; \} \\
= \; & \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists \sqsubseteq \in Marking(H') \colon \\
& \quad \preccurlyeq_{H'} \subseteq \sqsubseteq \; \wedge ReadPres(H', \sqsubseteq) \; \wedge \; WriteObs(H', \sqsubseteq)\} \\
= \; & Markable
\end{aligned}
$$

In these eight steps we apply:

1) the definition of $FinalStateOpaque$,

2) Lemma 51 and $S \in TComplete$ (because $H' \in TExtension(H)$ and $H' \equiv S$),

3) the definition of $FinalStateMarkable$ and $S \in TComplete$,

4) Lemma 48,

5) logical rearrangement,

6) transitivity of $\subseteq$,

7) Lemma 49, and

8) the definition of $FinalStateMarkable$. □

10.2.2 Marking TL2

Notation. Let us remind the notation. Consider an execution history H. We use $l_1 \prec_H l_2$ to denote that l_1 is executed before l_2. We use $l_1 \sim_H l_2$ to denote that l_1 is executed concurrently to l_2. We use $l_1 \precsim_H l_2$ to denote that l_1 is executed before or concurrently to l_2. We use $\prec_{clock}$, $\prec_{ver[i]}$ and $\prec_{lock[i]}$ to denote the linearization order of $clock$, $ver[i]$ and $lock[i]$ respectively.

A label c_1'c_2 is a call string that denotes a method call labeled c_2 that is executed in the body of the method call labeled c_1.

We use $initOf_H(T)$ and $commitOf_H(T)$ to denote the *init* and *commit* method calls of transaction T in history H.

For a TM algorithm specification π, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that π results.

Marking Relation. Now, we define the marking relation for TL2. The effect order of transactions is the linearization order of their calls to the *clock*. Every transaction reads an initial snapshot number at $I01$. A committing transaction makes a new snapshot at $C07$. A TL2 transaction takes effect at $C07$ if it is committed and at $I01$ otherwise. The access order of read operations and writer transactions to location i is the execution order of their accesses to the $reg[i]$ register. The read method reads $reg[i]$ at $R04$ and a writer transaction writes to $reg[i]$ at $C16_i$.

Definition 19 (Marking TL2)**.** *Consider an execution history $H \in \mathbb{H}(TL2)$. Let*

$$
\begin{aligned}
readAcc(R) &= R\text{'}R04 \\
writeAcc(T,i) &= commitOf_H(T)\text{'}C16_i \\
Eff(T) &= \begin{cases} initOf_H(T)\text{'}I01 & \text{if } T \in Aborted(H) \\ commitOf_H(T)\text{'}C07 & \text{if } T \in Committed(H) \end{cases}
\end{aligned}
$$

The marking $\sqsubseteq$ for H is the reflexive closure of $\sqsubset$ that is define as follows:

$$\{(T, T') \mid T, T' \in Trans(H) \ \wedge \ Eff(T) \prec_{clock} Eff(T')\} \cup$$
$$\{(T, R) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \ \wedge \ writeAcc(T, i) \prec_H readAcc(R)\} \cup$$
$$\{(R, T) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \ \wedge \ readAcc(R) \precsim_H writeAcc(T, i)\}$$

We have formally proved the markability of TL2 using a novel program logic that facilitates reasoning about execution and linearization orders. To keep the focus of this section on markability, we avoid the formal presentation of the logic and present a simplified reasoning.

Lemma 52. *TL2 preserves reads of aborted transactions (part 1).*

$$\forall H \in \mathbb{H}(TL2):$$
$$\forall R \in GlobalTReads(H):\ Let\ i = arg1_H(R), T = trans_H(R):$$
$$T \in Aborted(H) \Rightarrow NoWriterBetween_{H,i}(R, \sqsubseteq, T)$$

Proof Sketch.

T		T'	
		$C02_i \triangleright$	$lock[i].trylock()$
			...
		$\boxed{C07 \triangleright}$	$wver = clock.iaf()$
$\boxed{I01 \triangleright}$	$snap = clock.read()$		...
	...		
$R03 \triangleright$	$s_1 = ver[i].read()$		
$\boxed{R04 \triangleright}$	$v = reg[i].read()$		
		$\boxed{C16_i \triangleright}$	$reg[i].write(v)$
		$C17_i \triangleright$	$ver[i].write(wver)$
$R05 \triangleright$	$l = lock[i].read()$	$C18_i \triangleright$	$lock[i].unlock()$
$R06 \triangleright$	$s_2 = ver[i].read()$		
$R07 \triangleright$	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge\ s_1 = s_2 \ \wedge\ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 10.1: Case $T \in Aborted(H) \ \wedge\ R \sqsubset T' \sqsubset T$

We consider an aborted transaction T with an unaborted global read operation R from a location i and a writer T' of i.

We assume that

T' accesses i after R

that is

(1) $T' \sqsubset R$

and

T' takes effect before T

that is

(2) $T' \sqsubset T$

We show that

TL2 aborts R.

Figure 10.1 depicts the two transactions.

By Definition 19 on [1], we have

(3) $R04 \prec_H C16_i$

By Definition 19 on [2], we have

(4) $C07 \prec_{clock} I01$

The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.

Case 1:

(5) $R05 \prec_{lock[i]} C18_i$

By P2X on the algorithm, we have

(6) $C02_i \prec_H C07$

(7) $I01 \prec_H R05$

By the rule XLTrans on [6], [4] and [7], we have

$C02_i \prec_H R05$

thus, by the rule X2L, we have

(8) $C02_i \prec_{lock[i]} R05$

By the rule TryLockReadM on [8] and [5], we have that

$R05$ returns $true$ i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(9) $C18_i \prec_{lock[i]} R05$

By P2X on the algorithm, we have

(10) $C17_i \prec_H C18_i$

(11) $R05 \prec_H R06$

By the rule XLTrans on [10], [9] and [11], we have

$C17_i \prec_H R06$

Thus, by the rule X2L, we have

(12) $C17_i \prec_{ver[i]} R06$

By Lemma 61 on [12], we have

(13) $wver \leq s_2$

By P2X on the algorithm, we have

(14) $R03 \prec_H R04$

(15) $C16_i \prec_H C17_i$

By the rule XXTrans on [14], [3] and [15], we have

$R03 \prec_H C17_i$

Thus, by the rule X2L, we have

(16) $R03 \prec_{ver[i]} C17_i$

By Lemma 61 on [16], we have

(17) $s_1 < wver$

From [13] and [17], we have

$\neg(s_1 = s_2)$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 53. *TL2 preserves reads of aborted transactions (part 2).*

$$\forall H \in \mathbb{H}(TL2):$$
$$\forall R \in GlobalTReads(H):\ Let\ i = arg1_H(R), T = trans_H(R):$$
$$T \in Aborted(H) \Rightarrow NoWriterBetween_{H,i}(T, \sqsubseteq, R)$$

Proof Sketch.

T		T'	
$\boxed{I01 \triangleright}$	$snap = clock.read()$	$C02_i \triangleright$	$lock[i].trylock()$
$I02 \triangleright$	$rver[t].write(snap)$		
		$\boxed{C07 \triangleright}$	$wver = clock.iaf()$
		$\boxed{C16_i \triangleright}$	$reg[i].write(v)$
$\boxed{R04 \triangleright}$	$v = reg[i].read()$	$C17_i \triangleright$	$ver[i].write(wver)$
$R05 \triangleright$	$l = lock[i].read()$	$C18_i \triangleright$	$lock[i].unlock()$
$R06 \triangleright$	$s_2 = ver[i].read()$		
$R07 \triangleright$	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge\ s_1 = s_2 \ \wedge\ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 10.2: Case $T \in Aborted(H) \ \wedge\ T \sqsubset T' \sqsubset R$

We consider an aborted transaction T with an unaborted global read operation R from a location i and a writer T' of i.

We assume that

T' takes effect after T

that is

(1) $T \sqsubset T'$

and

T' accesses i before R

that is

(2) $T' \sqsubset R$

We show that

TL2 aborts R.

Figure 10.2 depicts the two transactions.

By Definition 19 on [1], we have

(3) $I01 \prec_{clock} C07$

By Definition 19 on [2], we have

(4) $C16_i \preceq R04$

The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.

Case 1:

(5) $R05 \prec_{lock[i]} C18_i$

By P2X on the algorithm, we have

(6) $C02_i \prec_H C16_i$

(7) $R04 \prec_H R05$

By the rule XXTrans on [6], [4] and [7], we have

$C02_i \prec_H R05$

thus, by the rule X2L, we have

(8) $C02_i \prec_{lock[i]} R05$

By the rule TryLockReadM on [8] and [5], we have that

$R05$ returns $true$ i.e. $l = true$.

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(9) $C18_i \prec_{lock[i]} R05$

By P2X on the algorithm, we have

(10) $C17_i \prec_H C18_i$

(11) $R05 \prec_H R06$

By the rule XLTrans on [10], [9] and [11], we have

$C17_i \prec_H R06$

Thus, by the rule X2L, we have

(12) $C17_i \prec_{ver[i]} R06$

By Lemma 60 on [12], we have

(13) $wver \leq s_2$

By the rule SCounter on [3], we have

(14) $snap < wver$

The value of $sver$ is read at $R07$ from $rver$.

The thread-local register $rver$ is only assigned at $I02$ to $snap$.

Thus, we have

(15) $snap = sver$

From [13], [14] and [15], we have

$sver > s_2$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 54. *TL2 preserves reads of aborted transactions.*

$$\forall H \in \mathbb{H}(TL2):$$
$$\forall R \in GlobalTReads(H): Let\ i = arg1_H(R), T = trans_H(R):$$
$$T \in Aborted(H) \Rightarrow$$
$$NoWriterBetween_{H,i}(R, \sqsubseteq, T)\ \wedge\ NoWriterBetween_{H,i}(T, \sqsubseteq, R)$$

Proof. Immediate from Lemma 52 and Lemma 53. □

Lemma 55. *TL2 preserves reads of committed transactions (part 1).*

$$\forall H \in \mathbb{H}(TL2):$$
$$\forall R \in GlobalTReads(H):\ Let\ i = arg1_H(R), T = trans_H(R):$$
$$T \in Committed(H) \Rightarrow$$
$$NoWriterBetween_{H,i}(R, \sqsubseteq, T)$$

Proof Sketch.

T		T'	
		$C02'_i \triangleright$	$lock[i].trylock()$
			...
$I01 \triangleright$	$snap = clock.read()$	$\boxed{C07' \triangleright}$	$wver' = clock.iaf()$
$I02 \triangleright$	$rver[t].write(snap)$		...
	...		
$\boxed{R04 \triangleright}$	$v = reg[i].read()$		
	...	$\boxed{C16'_i \triangleright}$	$reg[i].write(v')$
$\boxed{C07 \triangleright}$	$wver = clock.iaf()$		
	...		
$C08 \triangleright$	$sver = rver[t].read()$		
	if $(wver \neq sver + 1)$	$C17'_i \triangleright$	$ver[i].write(wver')$
$C10_i \triangleright$	$l = lock[i].read()$	$C18'_i \triangleright$	$lock[i].unlock()$
$C11_i \triangleright$	$s = ver[i].read()$		
	if $(\neg(\neg l \ \wedge\ s \leq sver))$ **foreach** $(j \in lset)$ $lock[j].unlock()$ **return** $\mathbb{A}$		

Figure 10.3: Case $T \in Committed(H)\ \wedge\ R \sqsubset T' \sqsubset T$

We consider a committed transaction T with an unaborted global read operation R from a location i and a writer T' of i.

We assume that

T' accesses i after R

that is

(1) $R \sqsubset T'$

and

T' takes effect before T

that is

(2) $T' \sqsubset T$

We show that

TL2 aborts R.

Figure 10.3 depicts the two transactions. We annotate the labels and variables of T' by a prime so that they do not conflict with the labels and variables of T.

By Definition 19 on [1], we have

(3) $R04 \prec_H C16_i$

By Definition 19 on [2], we have

(4) $C07' \prec_{clock} C07$

The method calls $I01$ and $C07'$ are on the object *clock*. We consider two cases for the linearization order of them.

Case 1:

(5) $C07' \prec_{clock} I01$

From [5] and [3],

The proof of this case reduces to the proof of Lemma 52.

Case 2:

(6) $I01 \prec_{clock} C07'$

By the rule SCounter on [4], we have

(7) $wver' < wver$

By the rule SCounter on [6], we have

(8) $snap < wver'$

The value of *sver* is read at $R07$ from *rver*.

The thread-local register *rver* is only assigned at $I02$ to *snap*.

Thus, we have

(9) $snap = sver$

From [8] and [9], we have

(10) $sver < wver'$

From [10] and [7], we have

(11) $wver \neq sver + 1$

Thus,

The if branch is taken.

The method calls $C10_i$ and $C18'_i$ are on the object $lock[i]$.

We consider two cases for the linearization order of them.

Case 2.1:

(12) $C10_i \prec_{lock[i]} C18'_i$

By P2X on the algorithm, we have

(13) $C02'_i \prec_H C07'$

(14) $C07 \prec_H C10_i$

By the rule XLTrans on [13], [4] and [14], we have

$C02'_i \prec_H C10_i$

thus, by the rule X2L, we have

(15) $C02'_i \prec_{lock[i]} C10_i$

By the rule TryLockReadM on [15] and [12], we have that

$R05$ returns $true$ i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2.2:

(16) $C18'_i \prec_{lock[i]} C10_i$

By P2X on the algorithm, we have

(17) $C17'_i \prec_H C18'_i$

(18) $C10_i \prec_H C11_i$

By the rule XLTrans on [17], [16] and [18], we have

$C17'_i \prec_H C11_i$

Thus, by the rule X2L, we have

(19) $C17'_i \prec_{ver[i]} C11_i$

By Lemma 61 on [19], we have

(20) $wver' \leq s$

From [10], [20], we have

$sver < s$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 56. *TL2 preserves reads of committed transactions (part 2).*

$$\forall H \in \mathbb{H}(TL2):$$
$$\forall R \in GlobalTReads(H): \text{Let } i = arg1_H(R), T = trans_H(R):$$
$$T \in Committed(H) \Rightarrow$$
$$NoWriterBetween_{H,i}(T, \sqsubseteq, R)$$

Proof Sketch.

T		T'	
$\boxed{R04 \triangleright}$	$v = reg[i].read()$		
	...		
$\boxed{C07 \triangleright}$	$wver = clock.iaf()$		
	...	$\boxed{C07' \triangleright}$	$wver' = clock.iaf()$
			...
		$\boxed{C16'_i \triangleright}$	$reg[i].write(v')$

Figure 10.4: Case $T \in Committed(H) \ \wedge \ T \sqsubset T' \sqsubset R$

We consider a committed transaction T with an unaborted global read operation R from a location i and a writer T' of i. We should show that it is impossible that T' takes effect after T and T' accesses i before R.

We assume that

T' takes effect after T

that is

(1) $T \sqsubset T'$

We show that

T' accesses i after R.

that is

(2) $R \sqsubset T'$

Figure 10.4 depicts the two transactions. We annotate the labels and variables of T' by a prime so that they do not conflict with the labels and variables of T.

By Definition 19 on [1], we have

(3) $C07 \prec_{clock} C07'$

By Definition 19 on [2], we have to show

$R04 \prec_H C16_i$

By P2X and the algorithm, we have

(4) $C04 \prec_H C07$

(5) $C07' \prec_H C16'_i$

By the rule XLTrans on [4], [3], and [5], we have

$R04 \prec_H C16_i$

□

Lemma 57. *TL2 preserves reads of committed transactions.*

$$\forall H \in \mathbb{H}(TL2)\colon$$
$$\forall R \in GlobalTReads(H)\colon Let\ i = arg1_H(R), T = trans_H(R)\colon$$
$$T \in Committed(H) \Rightarrow$$
$$NoWriterBetween_{H,i}(R, \sqsubseteq, T)\ \wedge\ NoWriterBetween_{H,i}(T, \sqsubseteq, R)$$

Proof. Immediate from Lemma 55 and Lemma 56. □

Lemma 58. *TL2 is read-preserving.*

$$\forall H \in \mathbb{H}(TL2)\colon ReadPres(H, \sqsubseteq)$$

Proof. Immediate from Lemma 54 and Lemma 57. □

Lemma 59. *Version registers are updated to ascending numbers.*

Let $C17_i^1$ *denote the method call at line* $C17_i$ *executed by a transaction* T_1 *and let* $wver^1$ *denote its argument. Similarly, let* $C17_i^2$ *denote the method call at line* $C17_i$ *executed by a transaction* T_2 *and let* $wver^2$ *denote its argument. If* $C17_i^1 \prec_{ver[i]} C17_i^2$*, then* $wver^1 < wver^2$*.*

Proof Sketch.

T_1		T_2	
	...		
$C02_i^1 \triangleright$	$locked^1 = lock[i].trylock()$		
	...		
$C07^1 \triangleright$	$wver^1 = clock.iaf()$		
	...		
$C17_i^1 \triangleright$	$ver[i].write(wver^1)$		
$C18_i^1 \triangleright$	$lock[i].unlock()$		...
	...	$C02_i^2 \triangleright$	$locked^2 = lock[i].trylock()$
			...
		$C07^2 \triangleright$	$wver^2 = clock.iaf()$
			...
		$C17_i^2$	$ver[i].write(wver^2)$
		$C18_i^2$	$lock[i].unlock()$
			...

Figure 10.5: Updating Version Registers

We have that

(1) $C17_i^1 \prec_{ver[i]} C17_i^2$

We show that

$wver^1 < wver^2$

By P2X on the algorithm, we have

(2) $C02_i^1 \prec_H C17_i^1$

(3) $C17_i^2 \prec_H C18_i^2$

By the rule XLTrans on [2], [1] and [3], we have

(4) $C02_i^1 \prec_H C18_i^2$

Thus, by the rule X2L, we have

(5) $C02_i^1 \prec_{lock[i]} C18_i^2$

From the algorithm,

(6) The ownership of $lock[i]$ is respected.

By the rule TryLock on [6] and [5], we have

(7) $C18_i^1 \prec_{lock[i]} C02_i^2$

By P2X on the algorithm, we have

(8) $C07_1 \prec_H C18_i^1$

(9) $C02_i^2 \prec_H C07^2$

By the rule XLTrans on [8], [7], and [9], we have

(10) $C07^1 \prec_H C07^2$

By the rule X2L on [10], we have

(11) $C07^1 \prec_{clock} C07^2$

By the rule SCounter on [11], we have

$wver^1 < wver^2$

□

Lemma 60. *For every write method call W on $ver[i]$ with argument v and every read method call R on $ver[i]$ with the return value v', if $W \prec_{ver[i]} R$ then $v \leq v'$.*

Proof Sketch.

We have

(1) W is a write method call on $ver[i]$.

(2) R is a read method call on $ver[i]$.

(3) $W \prec_{ver[i]} R$.

(4) The argument of W is v.

(5) The return value of R is v'.

We show that

$v \leq v'$

Let

(6) W' is last write on $ver[i]$ linearized before R.

(7) The argument of W' is v''.

By the rule AReg' on [6], [7], and [5], we have

(8) $v' = v''$

From [6], and [1], we have

(9) $W \preceq_{ver[i]} W'$

By the algorithm and [1], and [6], we have

(10) W and W' are both at $C17$.

By Lemma 59 on [10], [9], [4] and [7], we have

(11) $v \leq v''$

From [8] and [11], we have

$v \leq v'$

□

Lemma 61. *For every write method call W on $ver[i]$ with argument v and every read method call R on $ver[i]$ with the return value v', if $R \prec_{ver[i]} W$ then $v' < v$.*

Proof Sketch.

We have

(1) W is a write method call on $ver[i]$.

(2) R is a read method call on $ver[i]$.

(1) $R \prec_{ver[i]} W$.

(2) The argument of W is v.

(3) The return value of R is v'.

We show that

$v' < v$

Let

(4) W' is last write on $ver[i]$ linearized before R.

(7) The argument of W' is v''.

By the rule AReg' on [4], [7], and [3], we have

(8) $v' = v''$

From [1], and [4], we have

(9) $W' \prec_{ver[i]} W$

By the algorithm and [1], and [4], we have

(10) W and W' are both at $C17$.

By Lemma 59 on [10], [9], [4] and [7], we have

(11) $v'' < v$

From [8] and [11], we have

$v' < v$

□

Lemma 62. *TL2 is global-write-observant.*

$$\forall H \in \mathbb{H}(TL2)\colon$$
$$\forall R \in GlobalTReads(H)\colon \exists W \in GlobalTWrites(H)\colon Let\ T' = trans_H(W)\colon$$
$$LastPreAccessor_{H,\sqsubseteq}(T', R) \wedge$$
$$arg1_H(R) = arg1_H(W)\ \wedge\ retv_H(R) = arg2_H(W)$$

Proof Sketch.

We consider a transaction T with an unaborted global read operation R from a location i.

The read operation R is from the location i, thus,

(1) The argument of R is i.

As R is global, thus,

(2) The return value of R is the return value of $R04$.

We first show that

(3) The read method call from $reg[i]$ at $R04$ is race-free.

We assume that there is a write method call on $reg[i]$ concurrent to it and show that TL2 aborts R.

Figure 10.6 depicts this situation.

We assume that there a race between $R04$ and $C16_i$. Thus,

(4) $R04 \sim C16_i$

The method calls $R05$ and $C18_i$ are on the object $lock[i]$.

We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.

We consider two cases

Case 1:

(5) $R04 \prec_{lock[i]} C18_i$

By P2X and the algorithm, we have

T		T'	
		$C02_i \triangleright$	$locked = lock[i].trylock()$
			...
$R03 \triangleright$	$s_1 = ver[i].read()$		
$R04 \triangleright$	$v = reg[i].read()$	$C16_i \triangleright$	$v = reg[i].write(v)$
	...	$C17_i \triangleright$	$ver[i].write(wver)$
$R05 \triangleright$	$lock[i].read()$	$C18_i \triangleright$	$lock[i].unlock()$
$R06 \triangleright$	$s_2 = ver[i].read()$		...
$R07 \triangleright$	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 10.6: $R04$ is race-free

(6) $C02_i \prec_H C16_i$

(7) $R04 \prec_H R05$

By the rule XXTrans on [6], [4], and [7], we have

(8) $C02_i \prec_H R05$

By the rule X2L on [8], we have

(9) $C02_i \prec_{lock[i]} R05$

By the rule TryLockReadM on [9] and [5], we have that

$R05$ returns $true$ i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(10) $C18_i \prec_{lock[i]} R04$

By P2X and the algorithm, we have

(11) $R03 \prec_H R04$

(12) $R05 \prec_H R06$

(13) $C16_i \prec_H C17_i$

(14) $C17_i \prec_H C18_i$

By the rule XXTrans on [11], [4], and [13], we have

(15) $R03 \prec_H C17_i$

By Lemma 61 on [15], we have

(16) $s_1 < wver$

By the rule XLTrans on [14], [10], and [12], we have

(17) $C17_i \prec_H R06$

By Lemma 60 on [17], we have

(18) $s_2 > wver$

From [15] and [17], we have

(19) $s_1 \neq s_2$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Second, we show that

(20) The register $reg[i]$ is sequentially-written i.e. no two write methods on $reg[i]$ are concurrent.

We assume two concurrent write method calls on $reg[i]$ and show a contradiction.

Figure 10.7 depicts this situation.

T		T'	
$C02_i \triangleright$	$locked = lock[i].trylock()$		
	...	$C02'_i \triangleright$	$locked' = lock[i].trylock()$
			...
$C16_i \triangleright$	$v = reg[i].write(v)$	$C16'_i \triangleright$	$v' = reg[i].write(v')$
	...		...
$C18_i \triangleright$	$lock[i].unlock()$		
		$C18'_i \triangleright$	$lock[i].unlock()$

Figure 10.7: $reg[i]$ is sequentially-written

We assume that $C16_i$ and $C16'_i$ are concurrent. Thus,

(21) $C16_i \sim C16'_i$

By P2X and the algorithm, we have

(22) $C02_i \prec_H C16_i$

(23) $C16'_i \prec_H C18'_i$

By the rule XXTrans on [22], [21], and [23], we have

(24) $C02_i \prec_H C18'_i$

By the rule X2L on [8], we have

(25) $C02_i \prec_{lock[i]} C18'_i$

By the rule TryLock on [25], we have that

(26) $C18_i \prec_{lock[i]} C02'_i$

By P2X and the algorithm, we have

(27) $C16_i \prec_H C18_i$

(28) $C02'_i \prec_H C16'_i$

By the rule XLTrans on [27], [26], and [28], we have

(29) $C16_i \prec_H C16'_i$

That is a contradiction to [21].

By the rule BReg on [3], and [20], we have

(30) There is a write method call w on $reg[i]$ such that

The argument of w is equal to the return value of $R04$.

The last write method call on $reg[i]$ that is executed before $R04$ is w.

By the algorithm, we have

(31) The register $reg[i]$ is written only at $C16_i$.

From [28] and [29], we have

There is a transaction T' such that

(We annotate the labels and variables of T' by a prime

so that they do not conflict with the labels and variables of T.)

(32) The argument of $C16'_i$ is equal to the return value of $R04$.

(33) The last write method call on $reg[i]$ that is executed before $R04$ is $C16'_i$.

By the algorithm, we have

(34) The argument of $C16'_i$ is the value of the key i in the map $wset[T']$ in the commit.

(35) The map $wset[T']$ is updated only at $W01$ in a *write* of T' such that

The key is equal to the first argument of the *write*.

The value is equal to the second argument of the *write*.

From [34], and [35], we have

(36) There exists a write W of T'

(37) The first argument of W is equal to i.

(38) W is the last *write* of T' with the first argument equal to i.

(39) The second argument of W is equal to the argument of $C16'_i$.

From [1], and [37], we have

(40) The first argument of R is the first argument of W.

From [2], [32], and [39], we have

(41) The return value of R is the second argument of W.

From [38], we have

(42) W is a global write.

We show that

(43) The transaction T' is the last pre-accessor of R.

From [33], we have

(44) $C16'_i \prec_H R04$

By Definition 19 on [44], we have

(45) $T' \sqsubset R$

Now, we show that

(46) Every transaction T'' other than T' that accesses i before R, takes effect before T'.

We assume that

(47) $T'' \neq T'$

(48) $T'' \sqsubset R$

We should show that

$T'' \sqsubset T'$

By Definition 19 on [48], we have

(We annotate the labels and variables of T' by a double prime.)

(49) $C16''_i \prec_H R04$

From [33], [33], and [49], we have

(50) $C16''_i \prec_H C16'_i$

Consider Figure 10.8.

T''		T'	
$C02''_i \triangleright$	$locked'' = lock[i].tryLock()$		
	...	$C02'_i \triangleright$	$locked' = lock[i].tryLock()$
$C07'' \triangleright$	$wver'' = clock.iaf()$		...
	...	$C07' \triangleright$	$wver' = clock.iaf()$
$C16''_i \triangleright$	$reg[i].write(v'')$		...
	...	$C16'_i \triangleright$	$reg[i].write(v')$
$C18''_i \triangleright$	$lock[i].unlock()$		...
		$C18'_i \triangleright$	$lock[i].unlock()$

Figure 10.8: Effect-order of pre-accessors

By P2X and the algorithm, we have

(51) $C02''_i \prec_H C16''_i$

(52) $C16'_i \prec_H C18'_i$

By the rule XXTrans on [51], [50], and [52], we have

(53) $C02''_i \prec_H C18'_i$

By the rule X2L on [53], we have

(54) $C02''_i \prec_{lock[i]} C18'_i$

By the rule TryLock on [45], we have that

(55) $C18''_i \prec_{lock[i]} C02'_i$

By P2X and the algorithm, we have

(56) $C07''_i \prec_H C18''_i$

(57) $C02'_i \prec_H C07'_i$

By the rule XLTrans on [56], [55], and [57], we have

(58) $C07'' \prec_H C07'$

By Definition 19 on [58], we have

$T'' \sqsubset T'$.

The conclusion is

[36], [42], [40], [41], and [43]

□

Lemma 63. *TL2 is local-write-observant.*

$\forall H \in \mathbb{H}(TL2)$:

$\forall R \in LocalTReads(H)$: *Let* $T = trans_H(R), i = arg1_H(R), H' = H|T|i$:

$\exists W \in TWrites(H')$:

$$W \prec_{H'} R \ \wedge \ NoWriteBetween_{H'}(W, R) \wedge$$
$$retv_{H'}(R) = arg2_{H'}(W)$$

Proof Sketch.

Let

(1) The operation R is a local read with the first argument i by the transaction T.

From [1], as R is local, we have

(2) There is a write operation before R with the first argument i by T.

From [2], let

(3) The operation W is the last write operation before R with the first argument i by the

transaction T.

By the algorithm

(4) The value of a key i in $wset$ is updated only at $W01$ in a write operation with the first argument i

and the value of the key i is updated to the second argument of the write operation.

From [3] and [4], we have

(5) The value of a key i in $wset$ during the execution of R is equal to the second argument of W.

Thus, by the algorithm

(6) $R01$-$R02$ find a value for the key i in $wset$.

Thus,

(7) The return value of R is equal to the value of key i in $wset$.

From [7] and [5], we have

(8) The return value of R is equal to the second argument of W.

The conclusion is

[3] and [8]

□

Lemma 64. *TL2 is write-observant.*

$$\forall H \in \mathbb{H}(TL2)\colon WriteObs(H, \sqsubseteq)$$

Proof. Immediate from Lemma 63 and Lemma 62. □

Lemma 65. *TL2 is real-time-preserving.*

$$\forall H \in \mathbb{H}(TL2)\colon RealTimePres(H, \sqsubseteq)$$

Proof Sketch.

We assume that

(1) $T \preceq_H T'$

We show that

$T \sqsubseteq T'$

By the definition of $\preceq_H$, from [1], we have

(2) All the operations of T are executed before all the operations of T'.

By the rule X2L, from [2], we have

(3) All the operations of T on *clock* are linearized before all the operations of T' on *clock*.

By Definition 19,

(4) The effect point of each transaction is one of its own operations on the *clock* object.

From [3] and [4], we have

(5) The transaction T takes effect before the transaction T'.

that is

$T \sqsubseteq T'$

□

Lemma 66. *The relation $\sqsubseteq$ is a marking relation.*

$$\forall H \in \mathbb{H}(TL2)\colon \sqsubseteq \in Marking(H)$$

Proof Sketch.

Consider Definition 19.

By the totality of the linearization order $\prec_{clock}$, the relation $\sqsubseteq$ is a total on the set of transactions.

As every pair of method calls either execute in order or concurrently, every read operation of a location i is ordered either before or after every writer to i. In addition, as no method call can execute before another method call and also after after or concurrent to it, no read operation of a location i is ordered both before and after a writer to i.

□

Lemma 67. *TL2 is markable.*

$$\forall H \in \mathbb{H}(TL2) \colon H \in FinalStateMarkable$$

Proof. Immediate from Lemma 66, Lemma 58, Lemma 64, and Lemma 65.

□

Theorem 7. *TL2 is opaque.*

$$\forall H \in \mathbb{H}(TL2) \colon H \in FinalStateOpaque$$

Proof. Immediate from Lemma 67, and Theorem 1.

□

10.3 Testing TM Algorithms

10.3.1 Example: Dekker Mutual Exclusion

```
DekkerSpec {
   f_1: AtomicRegister
   f_2: AtomicRegister
   r: BasicRegister

   def this() {
      W_01>   f_1.write(0)
      W_02>   f_2.write(0)
   }

   main {
      {
         W_1>   f_1.write(1)
         R_2>   x_2 = f_2.read()
         I_1>   if (x_2 = 0)
         C_1>      r.write(1)
      } || {
         W_2>   f_2.write(1)
         R_1>   x_1 = f_1.read()
         I_2>   if (x_1 = 0)
         C_2>      r.write(2)
      }
   }

   order {
      W_1 -> R_2 &&
      W_2 -> R_1
   }

   spec {
      ~ (
         exec(C_1) /\
         exec(C_2)
      )
   }
}
```

Figure 10.9: Dekker Algorithm Specification

We introduce a DSL called Samand for the specification of concurrent object algorithms.

A specification of a concurrent object declares the type of a set of shared base objects and defines a set of methods. The set of supported base object types are `BasicRegister`, `AtomicRegister`, `AtomicCASRegister`, `Lock` and `TryLock`. There is also support for arrays of these types and thread-local objects. User can define record types. A record type contains a set of object declarations. The `new` operator dynamically allocates an instance of a record type and returns a reference to it. The method definitions call methods on the base objects. Method calls are ordered by program control and data dependencies and lock happens-before orders. To allow for performance benefits of out-of-order execution, method calls that are unordered by the program are allowed to appear reordered in the histories of the program. The user can explicitly require specific orders in the `order` block. Note that these orders can be translated to fences for specific architectures. In order to represent complete specifications, there is no implicit program order in the language. The language enforces the discipline that the object types and the program order are explicitly declared.

In the `main` block, the user can write a concurrent program that calls the methods of the specified concurrent object. The `main` block is a sequence of blocks, one for each thread. Finally, the `spec` block specifies the correctness assertion. Every history of the concurrent program is expected to satisfy the correctness assertion. The correctness assertion can assert a partial correctness condition. In particular, it can be the negation of a bug pattern.

The set of histories of a specification are constrained by two set of constraints. Firstly, every history respects the guarantees of the base objects. For example, if a base object is an atomic register, then the sub-history for that register should be linearizable. Secondly, every history respects the control, data and program order dependencies. For example, if a method call is data-dependent on another method call, then the latter should precede the former in the history.

Figure 10.9 shows the specification of Dekker mutual exclusion algorithm in Samand. The two flags are declared as atomic registers. The optional `this` method specifies the initialization statements. This method is executed before the concurrent execution begins.

```
1  W_01>  f_1.write(0)
2  W_02>  f_2.write(0)
3
4
5
6
7
8
9
10 .
```

```
1
2
3
4
5
6
7  W_1>  f_1.write(1)
8  R_2>  x_2 = f_2.read()
9  I_1>  if (x_2=1)
10 C_1>    r.write(1)
```

```
1
2
3  W_2>  f_2.write(1)
4  R_1>  x_1 = f_1.read()
5  I_2>  if (x_1=0)
6  C_2>    r.write(2)
7
8
9
10 .
```

Figure 10.10: Bug Trace for Incorrect If Condition

```
1  W_02>  f_2.write(0)
2  W_01>  f_1.write(0)
3
4
5
6
7
8
9
10 .
```

```
1
2
3  R_2>  x_2 = f_2.read()
4  I_1>  if (x_2=0)
5
6
7  W_1>  f_1.write(1)
8
9
10 C_1>      r.write(1)
```

```
1
2
3
4
5  W_2>  f_2.write(1)
6  R_1>  x_1 = f_1.read()
7
8  I_2>  if (x_1=0)
9  C_2>    r.write(2)
10 .
```

Figure 10.11: Bug Trace for Removed Program Order

This simple specification does not define any other method. The `main` block specifies the concurrent program. The `order` block specifies that each thread should set its own flag before reading the other thread's flag. Finally, the `spec` block specifies the correctness assertion i.e. the two critical sections should not both execute.

Running Samand checker on the specification of Dekker results in approval of the specification.

If the specification is not met, the Samand checker reports the trace that leads to violation of the specification in a graphical user interface. If the condition of the statement at line `I_1` is replaced with the incorrect condition `(x_2 = 1)`, Samand checker shows the interleaving depicted in Figure 10.10. If the declared order `W_1 -> R_2` is removed, Samand checker shows the interleaving depicted in Figure 10.11.

```
1 W_02>  f_2.write(0)
2 W_01>  f_1.write(0)
3
4
5
6
7
8 .
```

```
1
2
3 W_1>  f_1.write(1)
4 R_2>  x_2 = f_2.read()
5
6
7 I_1>  if (x_2=0)
8 C_1>    r.write(1)
```

```
1
2
3
4
5 W_2>  f_2.write(1)
6 R_1>  x_1 = f_1.read()
7
8 .
```

Figure 10.12: Dekker Random Execution

Samand checker is not only a checking tool but can also be viewed as an execution tool. The `false` literal is an assertion that any execution violates. Therefore, declaring `false` as the specification assertion results in a random execution. Updating the spec block of the dekker specification as follows shows an execution instance such as the execution depicted in Figure 10.12. In this execution only one of the critical sections `C_1` is executed.

```
spec {
   false
}
```

10.3.2 Language

The set of currently supported object types are basic registers `BasicRegister`, atomic registers `AtomicRegister`, atomic cas registers `AtomicCASRegister`, locks `Lock` and try-locks `TryLock`. As defined in the base objects section, atomic registers, atomic cas registers, locks and try-locks are linearizable objects and basic registers behave as registers only if they are not accessed concurrently.

A base object called `r` of type `BasicRegister` is declared as follows:

```
r: BasicRegister
```

There is also support for arrays. The following declaration declares an array of try-locks objects of size 10.

```
tryLocks: TryLock[10]
```

The 7th element of the array can be accessed by `tryLocks[6]`. There is also support for thread-local objects. A thread-local basic register can be declared as

```
reg: TLocal BasicRegister
```

Thread-local objects are arrays in nature. The thread identifier is implicitly passed for accesses to thread-local variables, and hence thread-local variables are conveniently accessed as normal objects. It is also possible to declare thread-local arrays. User-defined record types are also supported. For example, a `Node` type can be defined as follows:

```
Node {
   lock: Lock
   value: BasicRegister
   next: BasicRegister
}
```

A specification can declare methods. For instance, the following lines show the declaration of a transfer method.

```
def transfer(a) {
   L>     lock.lock()
   R1>    v1 = b1.read()
   R2>    v2 = b2.read()
   C1>    v3 = v1 - a
   C2>    v4 = v2 + a
   W1>    b1.write(v3)
   W2>    b2.write(v4)
   U>     lock.unlock()
   F>     return
}
```

Each method declaration has an implicit parameter for the calling thread identifier. The variable name `t` is reserved for this parameter and should not be used to name any other variable. The argument for this parameter is automatically passed at the call site. A statement is either a method call, a record creation, an if statement, a return statement or a math statement. The following statement allocates memory for an object of record type `Node` and returns a reference to it that is assigned to `ref`.

```
ref = new Node()
```

The following statement calls the method `method` on the base object `object` with the argument `arg` and assigns the return value to `ret`.

```
ret = object.method(arg)
```

If no receiver object is specified for a method call, the receiver is the current object. The following statement calls the method `method` on the field `object` of the record referenced by `ref` with the argument `arg` and assigns the return value to `ret`.

```
ret = ref.object.method(arg)
```

The supported math statements are of the form `x3 = x1 + x2` or `x3 = x1 - x2`.

The `main` block specifies the concurrent program. The thread blocks are separated by ||.

Data and control dependencies order method call. Correctness of the specified algorithm may be dependent on a specific order of method calls that are not ordered by data and control dependencies. The user can declare the required order of method calls in the `order` block. The program order of a specification is the transitive closure of data and control dependencies, the declared orders and the following conventional orders for locks and `this` method calls.

Locks (and try-locks) as the foundation of mainstream language memory models have ordering implications (in addition to the linearizability property). Every statement after a lock method (or a successful try-lock method) is ordered after it and every statement before an unlock method is ordered before it. Method calls on `this` object have ordering

implications as well. A function call whose side effects are not clear is even stronger than a compiler barrier. This excludes inline functions and functions known to be pure. We consider full ordering for method calls on `this` object. The statements before and after method calls on `this` object (and their enclosing statements) are ordered respectively before and after the call. In addition, the statements of `this` and `~this` methods are ordered respectively before and after all the statements of the concurrent program (the `main` block).

Finally, the `spec` block specifies the assertion that every history of the specification should satisfy. Note that the correctness assertion can assert complete or partial correctness of the specification such as the negation of a bug pattern. The assertion language supports conjunction `/\`, disjunction `\/`, negation `~` of assertions. Currently, atomic assertions can be that a specific method call is executed

```
exec(M)
```

a method call is executed before another method call

```
M1 \prec M2
```

an equality for variables and values

```
x1 = 2
x1 = x2
```

and `true` and `false` literals.

10.3.3 TM Algorithms in Samand

Note that we have restated the algorithms for the number of threads and locations that are needed for the testing program. Also the foreach loops and procedure callas are inlined. The set and map objects are implemented by registers.

The specification of DSTM is as follows:

```
Loc {
   writer: AtomicRegister
```

```
    oldValue: AtomicRegister
    newValue: AtomicRegister

    // These could be basic registers.
    // The bug exists even with these stronger registers.
}

DSTM {
    state: AtomicCASRegister[4]   // To store thread identifiers
    // Let state[3] be the state of the init trans
    start: AtomicCASRegister[2]   // To store Reference to Loc

    rset: TLocal AtomicRegister[2]
    // This could be a basic register array.
    // The bug exists even with these stronger registers.

    def this() {
        // init state and start
        I01>    state[1].write(\R)
        I02>    state[2].write(\R)
        I03>    state[3].write(\C)
        I04>    loc1 = new Loc()
        I05>    loc1.writer.write(3)
        I06>    loc1.newValue.write(0)
        I07>    start[0].write(loc1)
        I08>    loc2 = new Loc()
        I09>    loc2.writer.write(3)
        I10>    loc2.newValue.write(0)
        I11>    start[1].write(loc2)
```

```
}

def read(i) {
   R0>    s = state[t].read()
   R1>    if (s = \A)
   R2>       return \A
   R3>    start = start[i].read()
   // --------------
   // Stable value
   R4>    tp = start.writer.read()
   R5>    sp = state[tp].read()
   R6>    if (tp != t && sp = \R)
   R7>       state[tp].cas(\R, \A)
   R8>    if (sp = \A)
   R9>       v = start.oldValue.read()
          else
   R10>      v = start.newValue.read()
   // --------------
   R11>   if (tp != t)
   R12>      rset[i].write(v)
   // --------------
   // Validate
   R13>   rv0 = rset[0].read()
   R14>   if (rv0 != \bot) {
   R15>      s0 = start[0].read()
   R16>      wt0 = s0.writer.read()
   R17>      st0 = state[wt0].read()
   R18>      if (st0 = \C)
   R19>         vp0 = s0.newValue.read()
```

```
            else
   R20>        vp0 = s0.oldValue.read()
   R21>     if (rv0 != vp0)
   R22>        return \A
   R23>     cts0 = state[t].read()
   R24>     if (cts0 != \R)
   R25>        return \A
         }
   R26>  rv1 = rset[1].read()
   R27>  if (rv1 != \bot) {
   R28>     s1 = start[1].read()
   R29>     wt1 = s1.writer.read()
   R30>     st1 = state[wt1].read()
   R31>     if (st1 = \C)
   R32>        vp1 = s1.newValue.read()
            else
   R33>        vp1 = s1.oldValue.read()
   R34>     if (rv1 != vp1)
   R35>        return \A
   R36>     cts1 = state[t].read()
   R37>     if (cts1 != \R)
   R38>        return \A
         }
   // --------------
   R39>  return v
}

def write(i, v) {
   W0>    s = state[t].read()
```

```
   W1>    if (s = \A)
   W2>       return \A
   W3>    start = start[i].read()
   W4>    wt = start.writer.read()
   W5>    if (wt = t) {
   W6>       start.newValue.write(v)
   W7>       return \Ok
          }
   // --------------
   // Stable value
   W8>    tp = start.writer.read()
   W9>    sp = state[tp].read()
   W10>   if (tp != t && sp = \R)
   W11>      state[tp].cas(\R, \A)
   W12>   if (sp = \A)
   W13>      vp = start.oldValue.read()
          else
   W14>      vp = start.newValue.read()
   // --------------
   W15>   startp = new Loc()
   W16>   startp.writer.write(t)
   W17>   startp.oldValue.write(vp)
   W18>   startp.newValue.write(v)
   W19>   b = start[i].cas(start, startp)
   W20>   if (b = 1)
   W21>      return \Ok
          else
   W22>      return \A
}
```

```
def commit() {
   C01>   rv0 = rset[0].read()
   C02>   if (rv0 != \bot) {
   C03>      s0 = start[0].read()
   C04>      wt0 = s0.writer.read()
   C05>      st0 = state[wt0].read()
   C06>      if (st0 = \C)
   C07>         vp0 = s0.newValue.read()
             else
   C08>         vp0 = s0.oldValue.read()
   C09>      if (rv0 != vp0)
   C10>         return \A
   C11>      cts0 = state[t].read()
   C12>      if (cts0 != \R)
   C13>         return \A
          }
   C14>   rv1 = rset[1].read()
   C15>   if (rv1 != \bot) {
   C16>      s1 = start[1].read()
   C17>      wt1 = s1.writer.read()
   C18>      st1 = state[wt1].read()
   C19>      if (st1 = \C)
   C20>         vp1 = s1.newValue.read()
             else
   C21>         vp1 = s1.oldValue.read()
   C22>      if (rv1 != vp1)
   C23>         return \A
   C24>      cts1 = state[t].read()
```

```
   C25>       if (cts1 != \R)
   C26>          return \A
              }
   C27>    b = state[t].cas(\R, \C)
   C28>    if (b = 1)
   C29>       return \C
              else
   C30>       return \A
}

main {
   {
      S11>     rset[0].write(\bot)
      S12>     rset[1].write(\bot)

      L11>    v10 = read(0)
      L12>    v11 = read(1)
      L13>    write(0, 7)
      L14>    c1 = commit()
   } || {
      S21>     rset[0].write(\bot)
      S22>     rset[1].write(\bot)

      L21>    v20 = read(0)
      L22>    v21 = read(1)
      L23>    write(1, 7)
      L24>    c2 = commit()
   }
}
```

```
order {
   R3 -> R15 &&
   R3 -> R28 &&
   W16 -> W19 &&
   W17 -> W19 &&
   W18 -> W19 &&
   C02 -> C27 &&
   C15 -> C27
}

spec {
   ~(
      L11_Ret \prec L22_Inv /\
      L21_Ret \prec L12_Inv /\

      L12_Ret \prec L23_Inv /\
      L22_Ret \prec L13_Inv /\

      L13_Ret \prec L24_Inv /\
      L23_Ret \prec L14_Inv /\

      v10 = 0 /\
      v11 = 0 /\
      v20 = 0 /\
      v21 = 0 /\
      c1 = \C /\
      c2 = \C
   )
```

```
   }
}
```

The specification of McRT is as follows:

```
McRT {

   r: AtomicRegister[2]
   ver: AtomicRegister[2]
   lock: TryLock[2]

   rset: TLocal AtomicRegister[2]
   uset: TLocal AtomicRegister[2]
   // These regs could be basic register arrays

   def this() {
      L01>    lock[0].unlock()
      L02>    lock[1].unlock()
      L03>    r[0].write(0)
      L04>    r[1].write(0)
      L05>    ver[0].write(0)
      L06>    ver[1].write(0)
   }

   def read(i) {
      R0>    u = uset[i].read()
      R1>    if (u = \bot) {
      R2>       ve = ver[i].read()
      R3>       l = lock[i].read()
      R4>       if (l = 1) {
```

```
   R5>           ov0 = uset[0].read()
   R6>           if (ov0 != \bot) {
   R7>              r[0].write(ov0)
   R8>              lock[1].unlock()
                 }
   R9>           ov1 = uset[1].read()
   R10>          if (ov1 != \bot) {
   R11>             r[1].write(ov1)
   R12>             lock[1].unlock()
                 }
   R13>        return \A
            }
   R14>     r = rset[i].read()
   R15>     if (r = \bot)
   R16>        rset[i].write(ve)
         }
   R17>  v = r[i].read()
   R18>  return v
}

def write(i, v) {
   W0>   u = uset[i].read()
   W1>   if (u = \bot) {
   W2>      l = lock[i].tryLock()
   W3>      if (l = 0) {
   W4>         ov0 = uset[0].read()
   W5>         if (ov0 != \bot) {
   W6>            r[0].write(ov0)
   W7>            lock[0].unlock()
```

```
                }
   W8>          ov1 = uset[1].read()
   W9>          if (ov1 != \bot) {
   W10>            r[1].write(ov1)
   W11>            lock[1].unlock()
                }
   W12>       return \A
           }
   W13>    ov = r[i].read()
   W14>    uset[i].write(ov)
       }
   W15>  r[i].write(v)
   W16>  return \Ok
}

def commit() {
   C0>    ove0 = rset[0].read()
   C1>    if (ove0 != \bot) {
   C2>       l0 = lock[0].read()
   C3>       ve0 = ver[0].read()
   C4>       if ((l0 = 1)|| (ve0 != ove0)) {
   C5>          ov10 = uset[0].read()
   C6>          if (ov10 != \bot) {
   C7>             r[0].write(ov10)
   C8>             lock[0].unlock()
                }
   C9>          ov11 = uset[1].read()
   C10>         if (ov11 != \bot) {
   C11>            r[1].write(ov11)
```

```
C12>             lock[1].unlock()
                 }
C13>             return \A
             }
         }
C14>     ove1 = rset[1].read()
C15>     if (ove1 != \bot) {
C16>         l1 = lock[1].read()
C17>         ve1 = ver[1].read()
C18>         if ((l1 = 1) || (ve1 != ove1)) {
C19>             ov20 = uset[0].read()
C20>             if (ov20 != \bot) {
C21>                 r[0].write(ov20)
C22>                 lock[0].unlock()
                 }
C23>             ov21 = uset[1].read()
C24>             if (ov21 != \bot) {
C25>                 r[1].write(ov21)
C26>                 lock[1].unlock()
                 }
C27>             return \A
             }
         }

C28>     u0 = uset[0].read()
C29>     if (u0 != \bot) {
C30>         v0 = ver[0].read()
C31>         vp0 = v0 + 1
C32>         ver[0].write(vp0)
```

```
   C33>       lock[0].unlock()
          }
   C34>   u1 = uset[1].read()
   C35>   if (u1 != \bot) {
   C36>       v1 = ver[1].read()
   C37>       vp1 = v1 + 1
   C38>       ver[1].write(vp1)
   C39>       lock[1].unlock()
          }

   C40>   return \C
}

main {
   {
      I11>    rset[0].write(\bot)
      I12>    rset[1].write(\bot)
      I13>    uset[0].write(\bot)
      I14>    uset[1].write(\bot)

      L11>   r1 = read(1)
      L12>   write(0, 7)
      L13>   c1 = commit()
   } || {
      I21>    rset[0].write(\bot)
      I22>    rset[1].write(\bot)
      I23>    uset[0].write(\bot)
      I24>    uset[1].write(\bot)
```

```
        L21>    write(1, 7)
        L22>    r2 = read(0)
        L23>    c2 = commit()
      }
    }

    order {
      R2 -> R3 &&
      R3 -> R17 &&
      C2 -> C3 &&
      C16 -> C17
    }

    spec {
      ~(
        r1 = 7  /\
        r2 = 7  /\
        c1 = \A /\
        c2 = \A
      )
    }
}
```

10.4 Synchronization Object Program Logic

10.4.1 Soundness

Theorem 3 (Soundness).

$\forall \pi, \mathcal{A}\colon ((\pi, \Gamma \vdash \mathcal{A}) \ \wedge \ (\pi \models \Gamma)) \Rightarrow (\pi \models \mathcal{A})$.

Proof.

Hypothesis

(1) $\pi, \Gamma \vdash \mathcal{A}$

(2) $\mathcal{X} \models \Gamma$

Desired Conclusion

$\pi \models \mathcal{A}$

Let

(3) $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

(4) $\mathcal{D} = d^*$

(5) $\mathcal{P} = p_0, (p_1||p_2||...||p_n)$

(6) $\mathcal{X} = (X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

By Definitions 16, we need to show that

$\mathcal{X} \models \mathcal{A}$

Let

(7) $X' = \sigma(X)$

By definition [2.71] on [6] and [7], we have

(8) $X' \in \mathbb{H}(\pi)$

By definition [2.70] on [6], we have

(9) $\forall o\colon \mathcal{T}_{base}(o) \in BT \Rightarrow$
$X'|o \in \mathbb{H}_B(o)$

(10) $\forall o\colon \mathcal{T}_{base}(o) \in LT \Rightarrow$
$(X'|o, \mathcal{L}(o)) \in \mathbb{H}_L(o)$

$\exists X_1, ..., X_n\colon$

(11) $\forall i \in \{0..n\}\colon (X_i, \sigma) \in [\![p_i]\!] \wedge$

(12) $X'' \in Interleave(X_1, \ldots, X_n) \wedge$
$X = X_0 \cdot X''$

By definition [2.16] on [9], we have

(13) $\forall o\colon o \in \mathcal{T}_{base}(o) \in BT \Rightarrow$
$(X'|o \in Sequential) \Rightarrow$
$(X'|o \in SeqSpec(o))$

By definition [2.18] on [10], we have

(14) $\forall o\colon o \in \mathcal{T}_{base}(o) \in LT \Rightarrow$
$X'|o \equiv \mathcal{L}(o) \wedge$
$\mathcal{L}(o) \in SeqSpec(o) \wedge$
$\prec_{X'|o} \subseteq \prec_{\mathcal{L}(o)}$

Induction on the derivation of [1]:

Case rule X2L:

By rule X2L on [1], we have that

(15) $\mathcal{T}_{base}(o) \in LT$

(16) $\pi, \Gamma \vdash l \prec l'$

(17) $\pi, \Gamma \vdash obj(l) = obj(l') = o$

(18) $\mathcal{A} = l \prec_o l'$

We show that

$\mathcal{X} \models \mathcal{A}$

That is

$l \prec_{\mathcal{L}(\sigma(o))} l'$

By the induction hypothesis on [16] and [17],

and then [2], [6] and [7], we have

(19) $l \prec_{X'} l'$

(20) $obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$

From [19] and [20], we have

(21) $l \prec_{X'|\sigma(o)} l'$

By [15], we have

(22) $\mathcal{T}_{base}(\sigma(o)) \in LT$

By [10] and [22], we have

(23) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(o)$

By Lemma 8 on [23] and [21], we have

$l \prec_{\mathcal{L}(\sigma(o))} l'$

Case rule SRC:

We have that

(24) $\mathcal{A} = \bigvee_{i=1..n} c = c_i$

(25) $\pi, \Gamma \vdash exec(\varsigma\text{'}c)$

(26) $\pi, \Gamma \vdash obj(\varsigma\text{'}c) = \theta$

(27) $\pi, \Gamma \vdash name(\varsigma\text{'}c) = n$

(28) $Calls_\pi(basename(\theta), n) = \{\overline{c_i}\}$

We show that

$\mathcal{A} \models \bigvee_{i=1..n} c = c_i$

that is

$\bigvee_{i=1..n} c = c_i$

By the induction hypothesis on [25], [26], [27], and then [2], [6] and [7], we have

(29) $\varsigma\text{'}c \in X'$

(30) $obj_{X'}(\varsigma\text{'}c) = \varsigma\text{'}\theta$

(31) $name_{X'}(\varsigma\text{'}c) = n$

From [7] and [12] on [29], [30], [31], we have

$\exists i \in 0..n$:

(32) $\varsigma\text{'}c \in X_i$

(33) $obj_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}\theta$

(34) $name_{X_i}(\varsigma\text{'}c) = n$

By Lemma 69 on [11] and [32], we have

(35) $basename(obj_{X_i}(\varsigma\text{'}c)) = obj_\pi(c)$

(36) $name_{X_i}(\varsigma\text{'}c) = name_\pi(c)$

By the definition of *basename* and ', we have

(37) $basename(\varsigma\text{'}\theta) = basename(\theta)$

From [35], [30] and [37], we have

(38) $basename(obj_\pi(c)) = basename(\theta)$

From [36] and [34], we have

(39) $name_\pi(c) = n$

From the definition of $calls_\pi(basename(\theta), n)$ on [38] and [39], we have

(40) $c \in calls_\pi(basename(\theta), n)$

From [28] and [36], we have

$\bigvee_{i=1..n} c = c_i$

Case rule P2X:

We have that

(41) $\mathcal{A} = \varsigma\text{'}c_1 \preceq \varsigma\text{'}c_2$

(42) $c_1 \rightarrow_\pi c_2$

(43) $\pi, \Gamma \vdash exec(\varsigma\text{'}c_1)$

(44) $\pi, \Gamma \vdash exec(\varsigma\text{'}c_2)$

We show that

$\mathcal{X} \models \varsigma\text{'}c_1 \prec \varsigma\text{'}c_2$

that is

$\varsigma\text{'}c_1 \prec_{X'} \varsigma\text{'}c_2$

By the induction hypothesis on [43], [44], and then [2], we have

(45) $\mathcal{X} \models exec(\varsigma\text{'}c_1)$

(46) $\mathcal{X} \models exec(\varsigma\text{'}c_2)$

that is

(47) $\varsigma\text{'}c_1 \in X'$

(48) $\varsigma\text{'}c_2 \in X'$

From Lemma 73 on [8], [47] and [48] [42], we have

$\varsigma\text{'}c_1 \prec_{X'} \varsigma\text{'}c_2$

Case rule OX2IX:

We have that

(49) $\mathcal{A} = c_1\text{'}c_3 \prec c_2\text{'}c_4$

(50) $\pi, \Gamma \vdash c_1 \prec c_2$

(51) $\pi, \Gamma \vdash exec(c_1\text{'}c_3)$

(52) $\pi, \Gamma \vdash exec(c_2\text{'}c_4)$

We show that

$\mathcal{X} \models c_1\text{'}c_3 \prec c_2\text{'}c_4$

that is

$c_1\text{'}c_3 \prec_{X'} c_2\text{'}c_4$

By the induction hypothesis on [50], [51], [52], and then [2], we have

(53) $\mathcal{X} \models c_1 \prec c_2$

(54) $\mathcal{X} \models exec(c_1\text{'}c_3)$

(55) $\mathcal{X} \models exec(c_2\text{'}c_4)$

that is

(56) $c_1 \prec_{X'} c_2$

(57) $c_1\text{'}c_3 \in X'$

(58) $c_2\text{'}c_4 \in X'$

From [56], we have

(59) $rEv(c_1) \lhd_{X'} iEv(c_2)$

From Lemma 74 on [8] and [57], we have

(60) $rEv(c_1\text{'}c_3) \lhd_{X'} rEv(c_1))$

From Lemma 74 on [8], and [58], we have

(61) $(iEv(c_2) \lhd_{X'} iEv(c_2\text{'}c_4)$

From [60], [59] and [61], we have

(62) $rEv(c_1\text{'}c_3) \lhd_{X'} iEv(c_2\text{'}c_4)$

From [62], we have

(63) $c_1\text{'}c_3 \prec_{X'} c_2\text{'}c_4$

Case rule ICONTROL:

We have that

(64) $\mathcal{A} =$

$exec(c\text{'}c') \Leftrightarrow$

$exec(c) \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$c\text{'}cond_\pi(c') \wedge$

$\bigwedge_{i=1..n} \neg exec(c\text{'}c_i)$

(65) $Labels(name_\pi(c)) = \{\overline{c_i}\}$

(66) $PreReturns_\pi(c') = \{\overline{c_r}\}$

We show that

$\mathcal{X} \models \mathcal{A}$

That is

$c\text{'}c' \in X' \Leftrightarrow$

$c \in X' \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{c_r} \neg(c\text{'}c_r \in X')$

We first show that

$c\text{'}c' \in X' \Rightarrow$

$c \in X' \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

We assume that

(67) $c\text{'}c' \in X'$

We show that

$c \in X' \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

From [7] and [12] on [67], we have

$\exists i \in \{0..n\}$:

(68) $c\text{'}c' \in X_i$

By Lemma 70 on [65], [66], [11] and [68], we have

(69) $c \in X_i \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X_i)$

From [7] and [12] and uniqueness of label

c

on [69], we have

(70) $c \in X' \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

Now, we show that

$c \in X' \wedge$

$\bigvee_{c_i} c' = c_i \wedge$

$\sigma(c\text{'}cond_\pi(c')) \wedge$

$\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

$\Rightarrow$

$c\text{'}c' \in X'$

We assume that

(71) $c \in X' \wedge$

(72) $\bigvee_{c_i} c' = c_i \wedge$

(73) $\sigma(c\text{'}cond_\pi(c')) \wedge$

(74) $\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X')$

We show that

$c\text{'}c' \in X'$

From [7] and [12] on [71], we have

$\exists i \in \{0..n\}$:

(75) $c \in X_i$

From [7] and [12] on [74], we have

$\forall i \in \{0..n\}$:

(76) $\bigwedge_{i=1..n} \neg(c\text{'}c_i \in X_i)$

By Lemma 71 on [65], [66], [11], [75], [72], [73] and [76], we have

(77) $c\text{'}c' \in X_i$

From [7] and [12] on [77], we have

$c\text{'}c' \in X'$

Case rule OCONTROL:

Similar to rule ICONTROL using Lemma 72.

Case rule TSEQ:

We have that

(78) $\mathcal{A} = l_1 \prec l_2 \ \vee \ l_2 \prec l_1 \ \vee \ l_1 = l_2$

(79) $\pi, \Gamma \vdash exec(l_1)$

(80) $\pi, \Gamma \vdash exec(l_2)$

(81) $\pi, \Gamma \vdash thread(l_1) = thread(l_2)$

(82) $\pi, \Gamma \ \vdash \ obj(l_1) \ = \ obj(l_2) \ =$ **this** $\vee$

$(\neg obj(l_1) =$ **this** $\wedge \ \neg obj(l_2) =$ **this**$)$

We show that

$\mathcal{X} \models l_1 \prec l_2 \ \vee \ l_2 \prec l_1 \ \vee \ l_1 = l_2$

that is

$l_1 \prec_{X'} l_2 \ \vee \ l_2 \prec_{X'} l_1 \ \vee \ l_1 = l_2$

By the induction hypothesis on [79], [80], [81], [82], and then [2], we have

(83) $\mathcal{X} \models exec(l_1)$

(84) $\mathcal{X} \models exec(l_2)$

(85) $\mathcal{X} \models thread(l_1) = thread(l_2)$

(86) $\mathcal{X} \models obj(l_1) = obj(l_2) = \textbf{this} \ \vee$

$(\neg obj(l_1) = \textbf{this} \ \wedge \ \neg obj(l_2) = \textbf{this})$

that is

(87) $l_1 \in X'$

(88) $l_2 \in X'$

(89) $thread_{X'}(l_1) = thread_{X'}(l_2)$

(90) $obj_{X'}(l_1) = obj_{X'}(l_2) = \textbf{this} \ \vee$

$(\neg obj_{X'}(l_1) = \textbf{this} \ \wedge \ \neg obj_{X'}(l_2) = \textbf{this})$

By [11] and [12] on [87] and [88], we have

$\exists i, j \in 0..n$:

(91) $l_1 \in X_i \ \wedge \ (X_i, \sigma) \in [\![p_i]\!]$

(92) $l_2 \in X_j \ \wedge \ (X_j, \sigma) \in [\![p_j]\!]$

Case analysis on [90]:

Case

(93) $obj_{X'}(l_1) = obj_{X'}(l_2) = \textbf{this}$

By Lemma 75 on [8], [87], [88], [93], we have

$\exists c_1, c_2$:

(94) $l_1 = c_1$

(95) $l_2 = c_2$

By Lemma 77 on [91], [92], [94], [95], we have

(96) $thread_X(l_1) = T_i$

(97) $thread_X(l_2) = T_j$

From [96], [97] and [89], we have

(98) $i = j$

By Lemma 79 on [91], [92], and [94], [95],

and [98], we have

(99) $l_1 \prec_X l_2 \ \vee \ l_2 \prec_X l_1 \ \vee \ l_1 = l_2$

Case

(100) $\neg obj_{X'}(l_1) = \textbf{this} \ \wedge$

$\neg obj_{X'}(l_2) = \textbf{this}$

Similar to the previous case where lemmas 76, 78 and 80 are used.

Case rule TLocal:

We have that

(101) $\mathcal{A} = thread(l_1) = thread(l_2)$

(102) $\mathcal{T}(basename(\phi)) = ThreadLocal \ st$

(103) $\pi, \Gamma \vdash exec(l_1) \ \wedge \ exec(l_2)$

(104) $\pi, \Gamma \vdash obj(l_1) = obj(l_2) = \phi[u]$

We show that

$\mathcal{X} \models thread(l_1) = thread(l_2)$

that is

$thread_{X'}(l_1) = thread_{X'}(l_2)$

By the induction hypothesis on [104], and then [2], we have

(105) $\mathcal{X} \models exec(l_1) \ \wedge \ exec(l_2)$

(106) $\mathcal{X} \models obj(l_1) = obj(l_2) = \phi[u]$

that is

(107) $obj_{X'}(l_1) = obj_{X'}(l_2) = \phi[\sigma(u)]$

(108) $l_1 \in X'$

(109) $l_2 \in X'$

From [107] , we have

(110) $basename(obj_{X'}(l_1)) = \phi$

(111) $index(obj_{X'}(l_1)) = \sigma(u)$

(112) $basename(obj_{X'}(l_2)) = \phi$

(113) $index(obj_X(l_2)) = \sigma(u)$

From Lemma 81 on [3], [102], [8], [108] and

[110] we have

(114) $thread_{X'}(l_1) = index(obj_{X'}(l_1))$

From Lemma 81 on [3], [102], [8], [109] and

[112] we have

(115) $thread_{X'}(l_2) = index(obj_{X'}(l_2))$

From [114] and [111] we have

(116) $thread_{X'}(l_1) = \sigma(u)$

From [115] and [113] we have

(117) $thread_{X'}(l_2) = \sigma(u)$

From [116] and [117] we have

(118) $thread_{X'}(l_1) = thread_{X'}(l_2)$

Case rule Id:

We have that

(119) $\mathcal{A} = obj(\varsigma\text{'}c) = \varsigma\text{'}\theta \wedge$
$name(\varsigma\text{'}c) = n \wedge$
$thread(\varsigma\text{'}c) = \varsigma\text{'}\tau \wedge$
$arg^*(\varsigma\text{'}c) = \varsigma\text{'}u^* \wedge$
$retv(\varsigma\text{'}c) = \varsigma\text{'}x$

(120) $obj_\pi(c) = \theta$

(121) $name_\pi(c) = n$

(122) $thread_\pi(c) = \tau$

(123) $arg_\pi(c) = u$

(124) $retv_\pi(c) = x$

(125) $\pi, \Gamma \vdash exec(\varsigma\text{'}c)$

We show that

(126) $\mathcal{X} \models \mathcal{A}$

that is

$obj_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}\theta) \wedge$
$name_{X'}(\varsigma\text{'}c) = n \wedge$
$thread_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}\tau) \wedge$
$arg^*_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}u^*) \wedge$
$retv_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}x)$

By the induction hypothesis on [125], and then [2], we have

(127) $\mathcal{X} \models exec(\varsigma\text{'}c)$

that is

(128) $\varsigma\text{'}c \in X'$

From [7] and [128], we have

(129) $\varsigma\text{'}c \in X$

From [12] and [129], we have

$\exists i \in \{0..n\}$:

(130) $\varsigma\text{'}c \in X_i$

From Lemma 68 on [11] and [130], we have

(131) $obj_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}\theta \wedge$
$name_{X_i}(\varsigma\text{'}c) = n \wedge$
$thread_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}\tau \wedge$
$arg^*_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}u^* \wedge$
$retv_{X_i}(\varsigma\text{'}c) = \varsigma\text{'}x$

From [131], [12], we have

(132) $obj_X(\varsigma\text{'}c) = \varsigma\text{'}\theta \wedge$
$name_X(\varsigma\text{'}c) = n \wedge$
$thread_X(\varsigma\text{'}c) = \varsigma\text{'}\tau \wedge$
$arg^*_X(\varsigma\text{'}c) = \varsigma\text{'}u^* \wedge$
$retv_X(\varsigma\text{'}c) = \varsigma\text{'}x$

From [132], [7], we have

(133) $obj_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}\theta) \wedge$
$name_{X'}(\varsigma\text{'}c) = n \wedge$
$thread_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}\tau) \wedge$
$arg^*_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}u^*) \wedge$
$retv_{X'}(\varsigma\text{'}c) = \sigma(\varsigma\text{'}x)$

Case rule Caller:

We have that

(134) $\mathcal{A} =$
$c\text{'}t = thread(c) \wedge$
$c\text{'}x^* = arg^*(c) \wedge$
$\bigvee_{i=1..n}(exec(c\text{'}c_i) \wedge arg1(c\text{'}c_i) = retv(c))$

(135) $\pi, \Gamma \vdash exec(c)$

(136) $\pi, \Gamma \vdash obj(c) = \mathbf{this}$

(137) $\pi, \Gamma \vdash name(c) = n$

(138) $tpar_\pi(n) = t \ \wedge \ par1_\pi(n) = x$

(139) $Returns_\pi(n) = \{\overline{c_i}\}$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$\sigma(c\text{'}t) = thread_{X'}(c) \wedge$
$\sigma(c\text{'}x^*) = arg^*_{X'}(c) \wedge$
$\bigvee_{i=1..n}$
$(c\text{'}c_i \in X' \wedge$
$arg1_{X'}(c\text{'}c_i) = retv_{X'}(c))$

By induction hypothesis on [135], [136] and

[137], and then [2], [6] and [7], we have

(140) $c \in X'$

(141) $obj_{X'}(c) = \mathbf{this}$

(142) $name_{X'}(c) = n$

From [7] on [140], [141] and [142], we have

(143) $c \in X$

(144) $obj_X(c) = \mathbf{this}$

(145) $name_X(c) = n$

By Lemma 82 on [6], [138], [139], [143], [144], and [145], we have

(146) $\sigma(c\text{'}t) = \sigma(thread_X(c)) \wedge$

(147) $\sigma(c\text{'}x^*) = \sigma(arg^*_X(c)) \wedge$

(148) $\bigvee_{i=1..n}$

$(c\text{'}c_i \in X \wedge$

$\sigma(arg1_X(c\text{'}c_i)) = \sigma(retv_X(c)))$

From [7] on [146], [147], and [148], we have

$\sigma(c\text{'}t) = thread_{X'}(c) \wedge$

$\sigma(c\text{'}x^*) = arg^*_{X'}(c) \wedge$

$\bigvee_{i=1..n}$

$(c\text{'}c_i \in X' \wedge$

$arg1_{X'}(c\text{'}c_i) = retv_{X'}(c))$

Case rule RET:

We have that

(149) $tpar_\pi(n) = t \ \wedge \ par1_\pi(n) = x \quad n \wedge$

(150) $c' \in Returns_\pi(n)$

(151) $\pi, \Gamma \vdash exec(c\text{'}c')$

(152) $\mathcal{A} =$

$exec(c) \wedge$

$obj(c) = \textbf{this} \ \wedge \ name(c) = n \wedge$

$thread(c) = c\text{'}t \quad \wedge \quad arg^*(c) = c\text{'}x^* \wedge$

$retv(c) = arg1(c\text{'}c')$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$c \in X' \wedge$

$obj_{X'}(c) = \textbf{this} \ \wedge \ name_{X'}(c) = n \wedge$

$thread_{X'}(c) = \sigma(c\text{'}t) \wedge$

$arg^*_{X'}(c) = \sigma(c\text{'}x^*) \wedge$

$retv_{X'}(c) = arg1_{X'}(c\text{'}c')$

By induction hypothesis on [151], and then [2], [6] and [7], we have

(153) $c\text{'}c' \in X'$

From [7] and [153], we have

(154) $c\text{'}c' \in X$

From Lemma 84 on [6], [149], [150], and [154], we have

(155) $c \in X \wedge$

(156) $obj_X(c) = \textbf{this} \ \wedge \ name_X(c) =$

(157) $\sigma(thread_X(c)) = \sigma(c\text{'}t) \wedge$

(158) $\sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \wedge$

(159) $\sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))$

From [7] on [155]-[159], we have

$c \in X' \wedge$

$obj_{X'}(c) = \textbf{this} \ \wedge \ name_{X'}(c) = n \wedge$

$thread_{X'}(c) = \sigma(c\text{'}t) \wedge$

$arg^*_{X'}(c) = \sigma(c\text{'}x^*) \wedge$

$retv_{X'}(c) = arg1_{X'}(c\text{'}c')$

Case rule CALLEE:

Similar to rule RET

Case rule XASYM:

We have that

(160) $\pi, \Gamma \vdash l \prec l'$

(161) $\mathcal{A} =$

$\neg(l' \prec l) \ \wedge \ \neg(l' \sim l) \ \wedge \ \neg(l' = l)$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$\neg(l' \prec_{X'} l) \ \wedge \ \neg(l' \sim_{X'} l) \ \wedge \ \neg(l' = l)$

Straightforward from Lemma 1.

Case rule XTOTAL:

Straightforward from Lemma 4.

Case rule X2X:

Straightforward from Lemma 5.

Case rule LASYM:

We have that

(162) $\pi, \Gamma \vdash l \prec_o l'$

(163) $\mathcal{A} =$

$\neg(l' \prec_o l) \wedge$

$\neg(l' = l)$

We show that

$\mathcal{X} \models \mathcal{A}$

Let

(164) $O = \mathcal{L}(\sigma(o))$

We need to show that

$\neg(l' \prec_O l) \wedge$

$\neg(l' = l)$

Straightforward from Lemma 10.

Case rule LTOTAL:

We have that

(165) $\mathcal{T}_{base}(o) \in LT$

(166) $\pi, \Gamma \vdash exec(l) \ \wedge \ exec(l')$

(167) $\pi, \Gamma \vdash obj(l) = obj(l') = o$

(168) $\mathcal{A} = (l \prec_o l') \vee (l' \prec_o l) \vee (l' = l)$

We show that

$\mathcal{X} \models \mathcal{A}$

From [165], let

(169) $O = \mathcal{L}(\sigma(o))$

We need to show that

$(l \prec_O l') \ \vee \ (l' \prec_O l) \ \vee \ (l' = l)$

By induction hypothesis on [166] and [167],

and then [2], [6] and [7], we have

(170) $l \in X'$

(171) $l' \in X'$

(172) $obj_{X'}(l) = \sigma(o)$

(173) $obj_{X'}(l') = \sigma(o)$

From [172] and [173], we have

(174) $l \in X'|\sigma(o)$

(175) $l' \in X'|\sigma(o)$

From [165], we have

(176) $\mathcal{T}_{base}(\sigma(o)) \in LT$

From [10], and [176], we have

(177) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(o)$

By Lemma 12 on [177], [174], [175], we have

$$l \prec_O l' \ \vee \ l' \prec_O l \ \vee \ l' = l$$

Case rule L2X:

We have that

(178) $\pi, \Gamma \vdash l \prec_o l'$

(179) $\mathcal{A} = exec(l) \ \wedge \ exec(l') \wedge$
$obj(l) = obj(l') = o$

We show that

$$\mathcal{X} \models \mathcal{A}$$

that is

$$l \in X' \ \wedge \ l' \in X' \wedge$$
$$obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$$

Let

(180) $O = \mathcal{L}(\sigma(o))$

By induction hypothesis on [178], and then [2],

and [6], we have

(181) $l \prec_O l'$

From [10] on [180], we have

(182) $(X'|\sigma(o), \mathcal{L}(\sigma(o))) \in \mathbb{H}_L(\sigma(o))$

By Lemma 13 on [182] and [181], we have

$$l \in X' \ \wedge \ l' \in X'$$
$$obj_{X'}(l) = obj_{X'}(l') = \sigma(o)$$

Case rule XTrans:

Straightforward from Lemma 2.

Case rule XXTrans:

Straightforward from Lemma 3.

Case rule LTrans:

Straightforward from Lemma 11.

Case rule TReal:

We have that

(183) $\pi, \Gamma \vdash T \nprec T'$

(184) $\pi, \Gamma \vdash exec(l) \ \wedge \ thread(l) = T$

(185) $\pi, \Gamma \vdash exec(l') \ \wedge \ thread(l') = T'$

(186) $\mathcal{A} = l \prec l' \ \vee \ l = l'$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$l \prec_{X'} l'$

By induction hypothesis on [183], [184], and

[185], and then [2], [6] and [7],

we have

(187) $T \prec\!\!\prec_{X'} T'$

(188) $l \in X'$

(189) $thread_{X'}(l) = T$

(190) $l' \in X'$

(191) $thread_{X'}(l') = T'$

From [189], we have

(192) $l \in X'|T$

From [189], we have

(193) $l' \in X'|T'$

From [187], we have

(194) $\forall T, T' \colon X'|T \lhd_H X'|T'$

From [194], [192] and [193], we have

$l \prec_{X'} l'$

Case rule AReg:

We have that

(195) $\mathcal{T}_{base}(reg) = AtomicRegister$

(196) $\pi, \Gamma \vdash isRead_{reg}(l_R)$

(197) $\mathcal{A} = \exists \ell_W \colon$

$isWriter_{reg}(\ell_W, l_R) \wedge$

$retv(l_R) = arg1(\ell_W)$

Let

(198) $reg' = \sigma(reg)$

(199) $Reg = \mathcal{L}(reg')$

From [195] and [198], we have

(200) $reg' \in AtomicRegister$

From [10] and [200], [199], we have

(201) $(X'|reg', Reg) \in \mathbb{H}_L(reg')$

By the definition of $isWriter$ on [197], we have

(202) $\mathcal{A} = \exists \ell_W \colon$

$isWrite_{reg}(\ell_W) \wedge$

$\ell_W \prec_{reg} l_R \wedge$

$\forall \ell'_W \colon isWrite_{reg}(\ell'_W) \Rightarrow$

$(\ell'_W \preceq_{reg} \ell_W \ \vee \ l_R \prec_{reg} \ell'_W) \wedge$

$retv(l_R) = arg1(\ell_W)$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

$\exists l_W \colon$

$isXWrite_{X',reg'}(l_W) \wedge$

$l_W \prec_{Reg} l_R \wedge$

$\forall l'_W \colon isXWrite_{X',reg'}(l'_W) \Rightarrow$

$(l'_W \preceq_{Reg} l_W \ \vee \ l_R \preceq_{Reg} l'_W) \wedge$

$retv_{X'}(l_R) = arg1_{X'}(l_W)$

From [196], we have

(203) $\pi, \Gamma \vdash$
$exec(l_R) \wedge$
$obj(l_R) = reg \wedge$
$name(l_R) = read$

By induction hypothesis on [203], and then [2], [6] and [7], we have

(204) $l_R \in X' \wedge$
$obj_{X'}(l_R) = reg' \wedge$
$name_{X'}(l_R) = read$

From the definition of $isXRead$ on [204], we have

(205) $isXRead_{X',reg'}(l_R)$

By Lemma 17 on [200], [201] and [205], we have

(206) $\exists l_w$:
$isLWriter_{X'|reg',Reg,reg'}(l_W, l_R) \wedge$

$retv_{X'|reg'}(l_R) = arg1_{X'}(l_W)$

From the definition of $isLWriter$ on [206], we have

$\exists l_W$:
$isXWrite_{X'|reg',reg'}(l_W) \wedge$
$l_W \prec_{Reg} l_R \wedge$
$\forall l'_W : isXWrite_{X'|reg',reg'}(l'_W) \Rightarrow$
$(l'_W \preceq_{Reg} l_W \ \vee \ l_R \preceq_{Reg} l'_W) \wedge$
$retv_{X'|reg'}(l_R) = arg1_{X'|reg'}(l_W)$

After simplification, we have

$\exists l_W$:
$isXWrite_{X',reg'}(l_W) \wedge$
$l_W \prec_{Reg} l_R \wedge$
$\forall l'_W : isXWrite_{X',reg'}(l'_W) \Rightarrow$
$(l'_W \preceq_{Reg} l_W \ \vee \ l_R \preceq_{Reg} l'_W) \wedge$
$retv_{X'}(l_R) = arg1_{X'}(l_W)$

Case rule BREG:

Similar to rule AREG by Lemma 16.

Case rule CASREGREAD:

By Lemma 18.

Case rule CASREGCAST:

By Lemma 19.

Case rule CASREGCASF:

By Lemma 19.

Case rule LOCK:

We have that

(207) $\mathcal{T}_{base}(lo) = Lock$

(208) $\pi, \Gamma \vdash isOwnerRespecting(lo)$

(209) $\pi, \Gamma \vdash isLock_{lo}(l_{l_1})$

(210) $\pi, \Gamma \vdash isUnlock_{lo}(l_{u_2})$

(211) $\pi, \Gamma \vdash l_{l_1} \prec_{lo} l_{u_2}$

(212) $\mathcal{A} = \exists \ell_{u_1}, \ell_{l_2}:$

$isUnlock_{lo}(\ell_{u_1}) \wedge$

$thread(\ell_{u_1}) = thread(l_{l_1}) \wedge$

$isLock_{lo}(\ell_{l_2}) \wedge$

$thread(l_{u_2}) = thread(\ell_{l_2}) \wedge$

$\ell_{u_1} \prec_{lo} \ell_{l_2}$

Let

(213) $lo' = \sigma(lo)$

(214) $L = \mathcal{L}(lo')$

We show that

$\mathcal{X} \models \mathcal{A}$

that is

(215) $\exists l_{u_1}, l_{l_2}:$

$isXUnlock_{X',lo'}(l_{u_1}) \wedge$

$thread_{X'}(l_{l_1}) = thread_{X'}(l_{u_1}) \wedge$

$isXLock_{X',lo'}(l_{l_2}) \wedge$

$thread_{X'}(l_{l_2}) = thread_{X'}(l_{u_2}) \wedge$

$l_{u_1} \prec_L l_{l_2}$

By induction hypothesis on [208]-[211], and then [2], [6] and [7], we have

(216) $isXOwnerRespecting_{lo'}(X') \wedge$

(217) $isXLock_{X',lo'}(l_{l_1}) \wedge$

(218) $isXUnlock_{X',lo'}(l_{u_2}) \wedge$

(219) $l_{l_1} \prec_L l_{u_2})$

From [216]-[219], we have

(220) $isXOwnerRespecting_{lo'}(X'|lo') \wedge$

(221) $isXLock_{X'|lo',lo'}(l_{l_1}) \wedge$

(222) $isXUnlock_{X'|lo',lo'}(l_{u_2}) \wedge$

(223) $l_{l_1} \prec_L l_{u_2})$

From [207] and [213], we have

(224) $lo' \in Lock$

From Lemma 21 on [224], and [220]-[223],

we have

$\exists l_{u_1}, l_{l_2}:$

(225) $isXUnlock_{X'|lo',lo'}(l_{u_1}) \wedge$

(226) $thread_{X'|lo'}(l_{l_1}) = thread_{X'|lo'}(l_{u_1}) \wedge$

(227) $isXLock_{X'|lo',lo'}(l_{l_2}) \wedge$

(228) $thread_{X'|lo'}(l_{l_2}) = thread_{X'|lo'}(l_{u_2}) \wedge$

(229) $l_{u_1} \prec_L l_{l_2}$

From [225]-[229], we have

$\exists l_{u_1}, l_{l_2}:$

(230) $isXUnlock_{X',lo'}(l_{u_1}) \wedge$

(231) $thread_{X'}(l_{l_1}) = thread_{X'}(l_{u_1}) \wedge$

(232) $isXLock_{X',lo'}(l_{l_2}) \wedge$

(233) $thread_{X'}(l_{l_2}) = thread_{X'}(l_{u_2}) \wedge$

(234) $l_{u_1} \prec_L l_{l_2}$

Case rule LockReadL:

Similar to the proof of rule Lock using Lemma 22.

Case rule LockReadR:

Similar to the proof of rule Lock using Lemma 23.

Case rule TryLock:

Similar to the proof of rule Lock using Lemma 26.

Case rule TryLockReadL:

Similar to the proof of rule Lock using Lemma 27.

Case rule TryLockReadR:

Similar to the proof of rule Lock using Lemma 28.

Case rule SCounter:

By Lemma 30.

Case rule BasicSetContains:

By Lemma 31.

Case rule BasicSetAdd:

By Lemma 32.

Case rule BasicMapGet:

By Lemma 33.

Case rule BasicMapPut:

By Lemma 34.

The basic inference rules and the equivalence and arithmetic rules are standard. □

Lemma 68.

$\forall p, X, \sigma, \varsigma, c' :$

$((X, \sigma) \in [\![p]\!] \ \wedge \ \varsigma\text{'}c' \in X)$

$\Rightarrow$

$(obj_X(\varsigma\text{'}c') = \varsigma\text{'}obj_\pi(c') \ \wedge \ thread_X(\varsigma\text{'}c') = \varsigma\text{'}thread_\pi(c') \ \wedge$

$name_X(\varsigma\text{'}c') = name_\pi(c_i) \ \wedge \ arg1_X(\varsigma\text{'}c') = \varsigma\text{'}arg1_\pi(c') \ \wedge$

$retv_X(\varsigma\text{'}c') = \varsigma\text{'}retv_\pi(c'))$.

Proof.

Structural induction on p:

(1) Case $p = c \triangleright\ n_\tau(u^*){:}x$

Straightforward form definition [2.67].

(2) Case $p = p_1;\ p_2$

Straightforward form definition [2.68]

and

the induction hypothesis.

(3) Case $p = \textbf{if}\ b\ p_1\ \textbf{else}\ p_2$

Straightforward form definition [2.69]

and

the induction hypothesis. □

Lemma 69.

$\forall p, X, \sigma, \varsigma, c' :$

$((X, \sigma) \in [\![p]\!]\ \wedge\ \varsigma\text{'}c' \in X)$

$\Rightarrow$

$basename(obj_X(\varsigma\text{'}c')) = obj_\pi(c')\ \wedge\ name_X(\varsigma\text{'}c') = name_\pi(c').$

Proof.

Structural induction on p:

(1) Case $p = c \triangleright\ n_\tau(u^*){:}x$

Straightforward form definition [2.67]

and

$basename(c\text{'}obj_\pi(c')) = basename(obj_\pi(c'))$ and

the induction hypothesis.

(2) Case $p = p_1;\ p_2$

Straightforward form definition [2.68]

and

the induction hypothesis.

(3) Case $p = \textbf{if}\ b\ p_1\ \textbf{else}\ p_2$

Straightforward form definition [2.69]

and

the induction hypothesis. □

Lemma 70.

Let

$Labels(name_\pi(c)) = \{\overline{c_i}\}$

$PreReturns_\pi(c') = \{\overline{c_r}\}$

$\forall p, X, \sigma, c, c' :$

$$((X,\sigma) \in [\![p]\!] \ \wedge \ c\text{'}c' \in X)$$

$\Rightarrow$

$$c \in X \ \wedge \ \bigvee_{c_i} c' = c_i\sigma(c\text{'}cond_\pi(c')) \ \wedge \ \bigwedge_{c_r} \neg(c\text{'}c_r \in X).$$

Proof.

Structural induction on p:

(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$

Straightforward form definition [2.67]

(2) Case $p = p_1;\ p_2$

Straightforward form definition [2.68], the induction hypothesis and the uniqueness of label c.

(3) Case $p =$ **if** $b\ p_1$ **else** p_2

Straightforward form definition [2.69] and the induction hypothesis. □

Lemma 71.

Let

$$Labels(name_\pi(c)) = \{\overline{c_i}\}$$

$$PreReturns_\pi(c') = \{\overline{c_r}\}$$

$\forall p, X, \sigma, c, c'$:

$$((X,\sigma) \in [\![p]\!] \ \wedge$$

$$c \in X \ \wedge$$

$$\bigvee_{c_i} c' = c_i \ \wedge$$

$$\sigma(c\text{'}cond_\pi(c')) \ \wedge$$

$$\bigwedge_{c_r} \neg(c\text{'}c_r \in X))$$

$\Rightarrow$

$$c\text{'}c' \in X.$$

Proof.

Structural induction on p:

(1) Case $p = c \triangleright \ n_\tau(u^*){:}x$

Straightforward form definition [2.67]

(2) Case $p = p_1;\ p_2$

Straightforward form definition [2.68], and

the induction hypothesis.

(3) Case $p = \textbf{if } b\ p_1\ \textbf{else } p_2$

Straightforward form definition [2.69]

and

the induction hypothesis. □

Lemma 72.

Let

$\forall p, X, \sigma, c$:

$(X, \sigma) \in [\![p]\!]$

$\Rightarrow$

$\sigma(cond_\pi(c))$

$\Leftrightarrow$

$c \in X$.

Proof.

Structural induction on p:

(1) Case $p = c \triangleright\ n_\tau(u^*)$:$x$

Straightforward form definition [2.67]

$cond_\pi(c) = true$

(2) Case $p = p_1;\ p_2$

Straightforward form definition [2.68],

and

the induction hypothesis.

(3) Case $p = \textbf{if } b\ p_1\ \textbf{else } p_2$

Straightforward form definition [2.69]

and

the induction hypothesis.

$\sigma(b)$ for the the then part and

$\neg\sigma(b)$ for the else part. □

Lemma 73.

$\forall \pi, X, \varsigma, c_1, c_2$:

$X \in \mathbb{H}(\pi)\ \wedge$

$\varsigma\text{'}c_1 \in X\ \wedge$

$\varsigma\text{'}c_2 \in X\ \wedge$

$c_1 \rightarrow_\pi c_2$

$\Rightarrow$

$\varsigma' c_1 \prec_X \varsigma' c_2$.

Proof.

Case analysis on $c_1 \rightarrow_\pi c_2$

(1) Case: the initialization order

Straightforward form definition [2.71] and [2.70].

$X = X_0 \cdot X'$

(2) Case: the sequential order of the sequential programs p_i

Straightforward form structural induction on p_i

and definition [2.67], [2.68], and [2.69].

$X = X_1 \cdot X_2$

(3) Case: $\rightarrow_n$ of a method n.

Straightforward form definition [2.67]

$\forall c_i, c_j \in \{\overline{c_i}\}$:

$((c_i \rightarrow_n c_j) \ \wedge \ c'c_i \in X' \ \wedge \ c'c_j \in X') \Rightarrow$

$c'c_i \prec_{X'} c'c_j$ □

Lemma 74.

$\forall \pi, Xc, c'$:

$X \in \mathbb{H}(\pi) \ \wedge \ c'c' \in X) \Rightarrow$

$(iEv(c) \lhd_X iEv(c'c') \ \wedge \ rEv(c'c') \lhd_X rEv(c))$.

Proof.

We have that

(1) $X \in \mathbb{H}(\pi)$

(2) $c'c' \in X$

We show that

$iEv(c) \lhd_X iEv(c'c')$

$rEv(c'c') \lhd_X rEv(c)$

From definition 2.71 and [2.70]on [1] and [2], we have

$\exists X_i$:

(3) $(X_i, \sigma) \in [\![p_i]\!]$

(4) $c'c' \in X_i$

(5) $X_i \Subset X$

We show that

(6) $iEv(c) \lhd_{X_i} iEv(c'c')$

(7) $rEv(c'c') \lhd_{X_i} rEv(c)$

Structural induction on p:

(8) Case $p = c \triangleright\ n_\tau(u^*){:}x$

Straightforward form definition [2.67]

$X = inv(c \triangleright\ n_\tau(u)) \cdot X' \cdot ret(c \triangleright\ x')$

(9) Case $p = p_1;\ p_2$

Straightforward form definition [2.68],

the induction hypothesis and

the uniqueness of label c.

(10) Case $p = \textbf{if}\ b\ p_1\ \textbf{else}\ p_2$

Straightforward form definition [2.69]

and

the induction hypothesis.

From [5] on [6] and [7], we have

$iEv(c) \lhd_X iEv(c{'}c')$

$rEv(c{'}c') \lhd_X rEv(c)$

□

Lemma 75.

$\forall \pi, X, \sigma, c$:

$X \in \mathbb{H}(\pi)\ \wedge$

$l \in X\ \wedge$

$obj_X(l) = \textbf{\textit{this}}\ \wedge$

$\Rightarrow$

$\exists c{:}\ l = c.$

Proof.

From definition 2.71 and [2.70], we have

$\exists X_i$:

(1) $(X_i, \sigma) \in [\![p_i]\!]$

(2) $l \in X_i$

(3) $X_i \Subset X$

Straightforward form structural induction on

p_i □

Lemma 76.

$\forall \pi, X, \sigma, c$:

$X \in \mathbb{H}(\pi)\ \wedge$

$l \in X\ \wedge$

$\neg obj_X(l) = \textbf{\textit{this}}\ \wedge$

$\Rightarrow$

$\exists c, c' : l = c\text{'}c'.$

Proof. Similar to Lemma 75. □

Lemma 77.

$\forall \pi, \mathcal{T}, \mathcal{D}, \mathcal{P}, p_0, ..., p_n, X, \sigma, c:$

$(\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$

$\mathcal{P} = p_0, (p_1 || p_2 || ... || p_n) \ \wedge$

$(X, \sigma) \in [\![p_i]\!] \ \wedge$

$c \in X \ \wedge$

$\Rightarrow$

$thread_X(c) = i.$

Proof.

By structural induction on p_i, we have

(1) $c \in Labels(p_i)$

(2) $thread_X(c) = thread_\pi(c)$

From the well-formedness conditions, we have

The thread argument of each method call is the

identifier of the thread in which it is called.

(3) $\forall c \in Labels(p_i) : thread_\pi(c) = i$

From [1], [2] and [3], we have

(4) $thread_X(c) = T_i$ □

Lemma 78.

$\forall \pi, \mathcal{T}, \mathcal{D}, \mathcal{P}, p_0, ..., p_n, X, \sigma, c:$

$(\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$

$\mathcal{P} = p_0, (p_1 || p_2 || ... || p_n) \ \wedge$

$(X, \sigma) \in [\![p_i]\!] \ \wedge$

$c\text{'}c' \in X \ \wedge$

$\Rightarrow$

$\sigma(thread_X(c\text{'}c')) = i.$

Proof.

By structural induction on p_i, we have

$\exists n, \tau$:

(1) $c' \in Labels(n)$

(2) $thread_X(c\text{'}c') = c\text{'}thread_\pi(c')$

(3) $\sigma(c\text{'}tpar_\pi(n)) = \sigma(\tau)$

(4) $c \in X$

(5) $thread_X(c) = \tau$

By Lemma 77 on [4], we have

(6) $thread_X(c) = i$

From the well-formedness conditions, we have

The thread argument of each method call is the

identifier of the thread in which it is called.

(7) $\forall c' \in Labels(n)$: $thread_\pi(c') = tpar_\pi(n)$

From [2], [7], [3], [5], and [6], we have

(8) $\sigma(thread_X(c'c')) = i$ □

Lemma 79.

$\forall p, X, \sigma, c_1, c_2$:

$(X, \sigma) \in [\![p]\!] \ \wedge$

$c_1 \in X \ \wedge$

$c_2 \in X \ \wedge$

$\Rightarrow$

$c_1 \prec_X c_2 \ \vee \ c_2 \prec_X c_1 \ \vee \ c_1 = c_2$.

Proof. Straightforward structural induction on p. □

Lemma 80.

$\forall p, X, \sigma, c_1, c_2, c_3, c_4$:

$(X, \sigma) \in [\![p]\!] \ \wedge$

$c_1\text{'}c_2 \in X \ \wedge$

$c_3\text{'}c_4 \in X \ \wedge$

$\Rightarrow$

$c_1\text{'}c_2 \prec_X c_3\text{'}c_4 \ \vee \ c_3\text{'}c_4 \prec_X c_1\text{'}c_2 \ \vee \ c_1\text{'}c_2 = c_3\text{'}c_4$.

Proof. Straightforward structural induction on p. □

Lemma 81.

$\forall \pi, X, \phi, st$:

$\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P}) \ \wedge$

$\mathcal{T}(\phi) = Threadlocal\ st \ \wedge$

$X \in \mathbb{H}(\pi) \ \wedge$

$l \in X \ \wedge$

$basename(obj_X(l)) = \phi$

$\Rightarrow$

$thread_X(l)) = index(obj_X(l))$.

Proof.

We have

(1) $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

(2) $\mathcal{T}(\phi) = Threadlocal\ st$

(3) $X \in \mathbb{H}(\pi)$

(4) $l \in X$

(5) $basename(obj_X(l)) = \phi$

From definition 2.71 and 2.70 on [3] and [5], we have

$\exists X_i$:

(6) $l \in X_i$

(7) $(X_i, \sigma) \in [\![p_i]\!]$

(8) $basename(obj_{X_i}(l)) = \phi$

(9) $X_i \Subset X$

We show that

(10) $thread_{X_i}(l) = index(obj_{X_i}(l))$

Structural induction on p_i:

(11) Case $p_i = c \triangleright\ n_\tau(u^*){:}x$

Form definition [2.67], we have

(12) $l = c\text{'}c'$

(13) $index(object_{X_i}(c\text{'}c')) = c\text{'}index_\pi(c')$

(14) $thread_{X_i}(c\text{'}c') = c\text{'}thread_\pi(c')$

From the well-formedness conditions, we have

The thread argument of each method call is the

identifier of the thread in which it is called.

(15) $\forall c' \in Labels(n)\colon thread_\pi(c') = tpar_\pi(n)$

From the well-formedness conditions, we have

The array access index to every thread-local

object is the current thread identifier.

(16) $\forall \phi, st, c' :$

$\mathcal{T}(\phi) = Threadlocal\ st \wedge$

$c' \in Labels(n) \Rightarrow$

$index_\pi(c') = tpar_\pi(n)$

From [13], [14], [15], [16], we have

(17) $thread_{X_i}(l) = index(obj_{X_i}(l))$

(18) Case $p_i = p'\ p''$

Straightforward form definition [2.68],

the induction hypothesis and

the uniqueness of label l.

(19) Case $p = \textbf{if}\ b\ p_1\ \textbf{else}\ p_2$

Straightforward form definition [2.69] and

the induction hypothesis.

From [10] and [9], we have

$thread_X(l) = index(obj_X(l))$

□

Lemma 82.

$\forall \pi, X, \sigma, \mathcal{L}, c, n, t, x :$

$(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

$tpar_\pi(n) = t\ \wedge\ par1_\pi(n) = x$

$Returns_\pi(n) = \{\overline{c_i}\}$

$c \in X$

$obj_X(c) = \textbf{\textit{this}}$

$name_X(c) = n$

$\Rightarrow$

$\sigma(c\text{'}t) = thread_X(c)\ \wedge$

$\sigma(c\text{'}x^*) = arg^*_X(c)\ \wedge$

$\bigvee_{i=1..n}$

$(c\text{'}c_i \in X\ \wedge$

$arg1_X(c\text{'}c_i) = retv_X(c)).$

Proof.

We have that

(1) $(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

(2) $tpar_\pi(n) = t \ \wedge \ par1\pi(n) = x$

(3) $Returns_\pi(n) = \{\overline{c_i}\}$

(4) $c \in X$

(5) $obj_X(c) = \textbf{this}$

(6) $name_X(c) = n$

We show that

$\sigma(c\text{'}t) = \sigma(thread_X(c)) \wedge$

$\sigma(c\text{'}x^*) = \sigma(arg^*_X(c)) \wedge$

$\bigvee_{i=1..n}$

$(c\text{'}c_i \in X \wedge$

$\sigma(arg1_X(c\text{'}c_i)) = \sigma(retv_X(c)))$

From definition 2.70 on [1], [4], [5], and [6], we have

$\exists X_i$:

(7) $(X_i, \sigma) \in [\![p_i]\!]$

(8) $c \in X_i$

(9) $obj_{X_i}(c) = \textbf{this}$

(10) $name_{X_i}(c) = n$

(11) $X_i \Subset X$

We show that

(12) $\sigma(c\text{'}t) = \sigma(thread_{X_i}(c)) \wedge$

(13) $\sigma(c\text{'}x^*) = \sigma(arg^*_{X_i}(c)) \wedge$

(14) $\bigvee_{i=1..n}$

$(c\text{'}c_i \in X_i \wedge$

$\sigma(arg1_{X_i}(c\text{'}c_i)) = \sigma(retv_{X_i}(c)))$

Structural induction on p_i:

(15) Case $p_i = c \triangleright \ n_\tau(u^*)\text{:}x$

From the Well-formedness

condition of specifications that

Every branch of every method definition ends

in a return statement.

we have

$\exists c_r \in \{\overline{c_r}\}\text{:} \ \sigma(c\text{'}cond_\pi(c_i))$

The rest is straightforward form the following

conditions of definition [2.67]

$\forall c_i \in \{\overline{c_i}\}$:

$c\text{'}c_i \in X' \Leftrightarrow$

$(\sigma(c\text{'}cond_\pi(c_i)) \wedge$

$\forall c_j \in PreReturns_\pi(c_i) \Rightarrow \neg c\text{'}c_j \in X'$

and

$\forall c_r \in \{\overline{c_r}\}$:

$c\text{'}c_r \in X' \Rightarrow \sigma(x') = \sigma(c\text{'}arg1_\pi(c_r))$

(16) Case $p_i = p' \ p''$

Straightforward form definition [2.68],

the induction hypothesis and

the uniqueness of label c.

(17) Case $p = \textbf{if } b\ p_1\ \textbf{else } p_2$

Straightforward form definition [2.69]

and

the induction hypothesis.

From [11] on [12], [13] and [14], we have

$\sigma(c\text{'}t) = \sigma(thread_X(c)) \wedge$

$\sigma(c\text{'}x^*) = \sigma(arg^*_X(c)) \wedge$

$\bigvee_{i=1..n}$

$(c\text{'}c_i \in X \wedge$

$\sigma(arg1_X(c\text{'}c_i)) = \sigma(retv_X(c)))$ □

Lemma 83.

$\forall X, \sigma, c, n, \tau, u, x' :$

$(X, \sigma) \in [\![c \triangleright\ n_\tau(u){:}x]\!]$

$c', c'' \in Returns_\pi(n)$

$c\text{'}c' \in X\ \wedge\ c\text{'}c'' \in X$

$\Rightarrow$

$c' = c''.$

Proof.

We have that

(1) $(X, \sigma) \in [\![c \triangleright\ n_\tau(u){:}x]\!]$

(3) $c' \in Returns_\pi(n)$

(3) $c'' \in Returns_\pi(n)$

(4) $c\text{'}c' \in X$

(5) $c\text{'}c'' \in X$

We show that

$c' = c''$

We consider three cases

Case

$c' = c''$

Obvious

Case

$c' \in PreReturns_\pi(c'')$

By definition [2.67] on [5], we have

$\neg c\text{'}c' \in X$

which is contradiction to [4].

Case

$c'' \in PreReturns_\pi(c')$

By definition [2.67] on [4], we have

$\neg c\text{'}c'' \in X$

which is contradiction to [5]. □

Lemma 84.

$\forall \pi, X, \sigma, \mathcal{L}, c, c', n, t, x$:

$(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

$tpar_\pi(n) = t \ \wedge \ par1_\pi(n) = x$

$c' \in Returns_\pi(n)$

c'$c' \in X$

$\Rightarrow$

$c \in X \ \wedge$

$obj_X(c) = \textbf{\textit{this}} \ \wedge \ name_X(c) = n \ \wedge$

$\sigma(thread_X(c)) = \sigma(c\text{'}t) \ \wedge$

$\sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \ \wedge$

$\sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))$.

Proof.

We have that

(1) $(X, \sigma, \mathcal{L}) \in [\![\pi]\!]$

(2) $tpar_\pi(n) = t \ \wedge \ par1\pi(n) = x$

(3) $c' \in Returns_\pi(n)$

(4) c'$c' \in X$

We show that

$c \in X \wedge$

$obj_X(c) = \textbf{this} \ \wedge \ name_X(c) = n \wedge$

$\sigma(thread_X(c)) = \sigma(c\text{'}t) \wedge$

$\sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \wedge$

$\sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))$

From definition 2.70 on [1] and [4], we have

$\exists X_i$:

(5) $(X_i, \sigma) \in [\![p_i]\!]$

(6) c'$c' \in X_i$

(7) $X_i \Subset X$

We show that

(8) $c \in X_i \wedge$

(9) $obj_{X_i}(c) = \textbf{this} \wedge name_{X_i}(c) = n \wedge$

(10) $\sigma(thread_{X_i}(c)) = \sigma(c\text{'}t) \wedge$

(11) $\sigma(arg^*_{X_i}(c)) = \sigma(c\text{'}x^*) \wedge$

(12) $\sigma(retv_{X_i}(c)) = \sigma(arg1_{X_i}(c\text{'}c'))$

Structural induction on p_i:

(13) Case $p_i = c \triangleright \ n_\tau(u^*)$:$x$

Straightforward form definition [2.67]

and

Lemma 83.

(14) Case $p_i = p' \ p''$

Straightforward form definition [2.68], the induction hypothesis and the uniqueness of label c.

(15) Case $p = \textbf{if } b\ p_1\ \textbf{else } p_2$

Straightforward form definition [2.69] and the induction hypothesis.

From [11] on [8]-[12], we have

$$
\begin{aligned}
& c \in X \wedge \\
& obj_X(c) = \textbf{this} \ \wedge \ name_X(c) = n \wedge \\
& \sigma(thread_X(c)) = \sigma(c\text{'}t) \wedge \\
& \sigma(arg^*_X(c)) = \sigma(c\text{'}x^*) \wedge \\
& \sigma(retv_X(c)) = \sigma(arg1_X(c\text{'}c'))
\end{aligned}
$$

□

10.4.2 Derived Rules

P2L:

Derived from rule P2X and rule X2L.

IX2OX:

Derived from rule X2X, rule CALLEE, rule TSEQ, rule OX2IX, and rule XASYM.

XLTRANS:

Derived from rule L2X, rule XTOTAL, rule XTRANS, rule XXTRANS, rule X2L, and rule LASYM.

X2L':

Derived from rule L2X, rule XTOTAL, rule X2L, and rule LASYM.

AREG':

Derived from rule AREG and the following

$$(\pi, \Gamma \vdash isWriter_{reg}(l_W, l_R) \ \wedge \ isWriter_{reg}(l_{W'}, l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_{W'})$$

BReg':

Derived from rule BReg and the following

$$\pi, \Gamma \vdash isSequential(reg) \Rightarrow \pi, \Gamma \vdash \forall \ell \colon (isRead_{reg}(\ell) \ \vee \ isWrite_{reg}(\ell)) \Rightarrow isRaceFree_{reg}(\ell)$$

TReg:

Derived from rule TLocal, rule TSeq and rule BReg'.

CASRegRead':

Derived from rule CASRegRead and the following

$$(\pi, \Gamma \vdash isCWriter_{reg}(l_W, l_R) \ \wedge \ isCWriter_{reg}(l_{W'}, l_R)) \Rightarrow (\pi, \Gamma \vdash l_W = l_{W'})$$

SCounter':

Derived from rule LTotal and rule SCounter.

BasicMapGet':

Derived from rule BasicMapGet.

BasicMapPut':

Derived from rule BasicMapPut.

DisjSyllL:

Derived form rule DisjElim and rule NegElim.

DisjSyllR:

Derived form rule DisjElim and rule NegElim.

CondElim':

Derived form rule Premise, rule CondElim, and rule NegIntro.

Other Lemmas:

Lemma 36:

Derived from rule Premise.

Lemma 37:

Derived from rule Premise.

10.5 Syntactic TM Correctness

10.5.1 Transactions

Let us define

$$Inits(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = init\} \tag{10.3}$$

$$Reads(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = read\} \tag{10.4}$$

$$Writes(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = write\} \tag{10.5}$$

$$Commits(X) = \{l \mid l \in X \ \wedge \ obj_X(l) = this \ \wedge \ name_X(l) = commit\} \tag{10.6}$$

$$Committed(X) = \{T \mid \exists l\colon l \in Commits(X) \ \wedge \ thread_X(l) = T \ \wedge \ retv_X(l) = \mathbb{C}\} \tag{10.7}$$

$$Aborted(X) = \{T \mid \exists l\colon l \in X \ \wedge \ obj_X(l) = this \ \wedge \ thread_X(l) = T \ \wedge \ retv_X(l) = \mathbb{A}\} \tag{10.8}$$

Lemma 85.

$\forall X, \sigma, c$:

$(X, \sigma) \in [\![trans_j]\!] \ \wedge$

$c \in X$

$\Rightarrow$

$(c \in Inits(X) \ \wedge \ c = IL_j \ \vee$

$c \in Reads(X) \ \vee$

$c \in Writes(X) \ \vee$

$c \in Commits(X) \ \wedge \ c = CL_j) \ \wedge$

$(IL_j \preceq c) \ \wedge$

$(CL_j \in X \Rightarrow c \preceq CL_j)$

Proof.

Case $j = 0$:

Derived from Equation 2.3 and Equation 2.68.

Case $0 < j \leq n$:

Derived from Equation 2.4, induction on the structure of *op* and Equation 2.69. □

Lemma 86.

$\forall X, \sigma$:

$(X, \sigma) \in [\![trans_j]\!]$

$\Rightarrow$

$\exists c$:

$c \in X \ \wedge \ obj_X(c) = this \ \wedge \ thread_X(c) = j \ \wedge$

$(retv_X(c) = \mathbb{C} \ \vee \ retv_X(c) = \mathbb{A})$

Proof.

Case $j = 0$:

Derived from Equation 2.3, Equation 2.68, Equation 2.67 and the well-formedness condition

$\forall c' \in Returns_\pi(commit)$: $retv_\pi(c') = \mathbb{C} \ \vee \ retv_\pi(c') = \mathbb{A}$.

Case $0 < j \leq n$:

Derived from Equation 2.4, induction on the structure of *op* and Equation 2.69, Equation 2.67 and the well-formedness condition

$\forall c' \in Returns_\pi(commit)$: $retv_\pi(c') = \mathbb{C} \ \vee \ retv_\pi(c') = \mathbb{A}$. □

Lemma 87.

$\forall X, \sigma, c, c'$:

$(X, \sigma) \in [\![trans_j]\!]$

$c \in X \ \wedge \ obj_X(c) = this \ \wedge \ thread_X(c) = j \ \wedge$

$c' \in X \ \wedge \ obj_X(c') = this \ \wedge \ thread_X(c') = j \ \wedge$

$(retv_X(c) = \mathbb{C} \ \vee \ retv_X(c) = \mathbb{C}) \ \vee \ (retv_X(c') = \mathbb{A} \ \vee \ retv_X(c') = \mathbb{A}) \Rightarrow$

$c = c'$

Proof.

Case $j = 0$:

Derived from Equation 2.3, Equation 2.68, Equation 2.67 and the well-formedness conditions

$\forall c \in Returns_\pi(init)\colon arg1_\pi(c) = ok$

$\forall c \in Returns_\pi(write)\colon arg1_\pi(c) \neq \mathbb{C}$

and that in every execution of the transaction $trans_0$, all the *write* method calls return *ok*.

Case $0 < j \leq n$:

Derived from Equation 2.4, induction on the structure of *op* and Equation 2.69, Equation 2.67 and

the following well-formedness conditions

$\forall c \in Returns_\pi(init)\colon arg1_\pi(c) = ok$

$\forall c \in Returns_\pi(read)\colon arg1_\pi(c) \neq \mathbb{C}$

$\forall c \in Returns_\pi(write)\colon arg1_\pi(c) \neq \mathbb{C}$

$\forall c \in Returns_\pi(commit)\colon arg1_\pi(c) = \mathbb{C} \ \vee \ arg1_\pi(c) = \mathbb{A}$ □

Lemma 88.

$\forall \pi \in \Pi_{TM}\colon \forall X \in \mathbb{H}(\pi)\colon \forall T \in Trans(X)\colon Let\ l = commitOf(T)\colon l \in Inits(X) \ \wedge \ thread_X(l) = T$

Proof. Derived from Equation 2.3, Equation 2.4, Equation 2.5, Equation 2.71, Equation 2.70, and Equation 2.68. □

Lemma 89.

$\forall \pi \in \Pi_{TM}\colon \forall X \in \mathbb{H}(\pi)\colon \forall l, l'\colon$

$(l \in Inits(X) \ \wedge \ l' \in Inits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$

$l = l'$

Proof. Derived from Equation 2.71, Equation 2.70, Lemma 75, Lemma 77, and Lemma 85.

□

Lemma 90.

$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$

$(l \in Inits(X) \ \wedge \ l' \in X \ \wedge \ obj_X(l') = this \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$

$l \preceq_X l'$

Proof. Derived from Equation 2.71, Equation 2.70, Lemma 75, Lemma 77, and Lemma 85.

□

Lemma 91.

$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall T \in Trans(X)$

$Let \ l = commitOf(T) :$

$T \in Committed(X) \Rightarrow$

$(l \in Commits(X) \ \wedge \ thread_X(l) = T)$

Proof. Derived from Equation 2.6, Equation 2.71, Equation 2.70, Lemma 75, Lemma 85 and Lemma 77. □

Lemma 92.

$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$

$(l \in Commits(X) \ \wedge \ l' \in Commits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$

$l = l'$

Proof. Derived from Equation 2.71, Equation 2.70, Lemma 75, Lemma 77 and Lemma 85.

□

Lemma 93.

$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall l, l' :$

$(l \in X \ \wedge \ obj_X(l) = this \ \wedge \ l' \in Commits(X) \ \wedge \ thread_X(l) = thread_X(l')) \Rightarrow$

$l \preceq_X l'$

Proof. Derived from Equation 2.71, Equation 2.70, Lemma 75, Lemma 77 and Lemma 85.

□

Lemma 94.

$\forall \pi \in \Pi_{TM} : \forall X \in \mathbb{H}(\pi) : \forall t : 0 \leq t \leq n$

$$(t \in Committed(X) \ \wedge \ t \in \neg Aborted(X)) \ \vee \ (t \in Aborted(X) \ \wedge \ t \in \neg Committed(X))$$

Proof. Derived from Equation 2.71, Equation 2.70, Lemma 86, and Lemma 87. □

Lemma 39

$\forall \pi \in \Pi_{TM} : \pi \models \Gamma_0$.

Proof. Derived from Equations 10.3-10.8, Equations 6.7-6.13, Definition 15, Definition 16 and Lemmas 88-94. □

Theorem 40.

Proof. Derived from Theorem 3 and Lemma 39. □

10.5.2 Markability

Theorem 5:

$Markable \subseteq Opaque$.

Proof.

We assume that

(1) $\pi \in Markable$

We show that that

$\pi \in Opaque$

By Definition 17 on [1], we have

(2) $\pi, \Gamma_0 \vdash isMarking(\sqsubseteq)$

From Lemma 40 on [2], we have

(3) $\pi \models isMarking(\sqsubseteq)$

By Definition 15, and Figure 6.2, Figure 3.5

on [3], we have

(4) $\forall \mathcal{X} \in [\![\pi]\!] \colon X \in FinalStateMarkable$

By Theorem 1 on [4], we have

(5) $\forall \mathcal{X} \in [\![\pi]\!] \colon X \in FinalStateOpaque$

By Definition 14 on [5], we have

(6) $\pi \in Opaque$ □

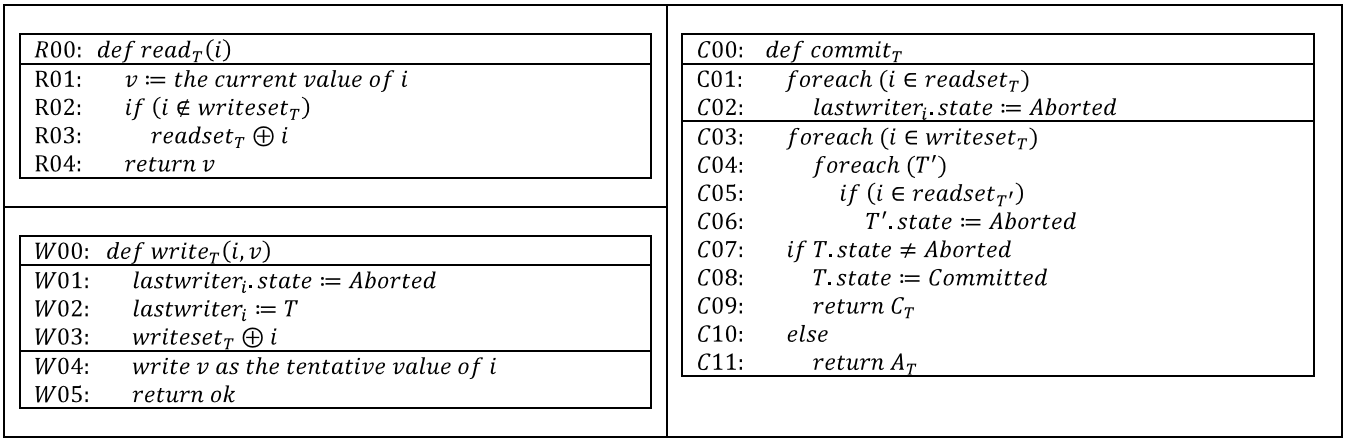


Figure 10.13: Specification of DSTM

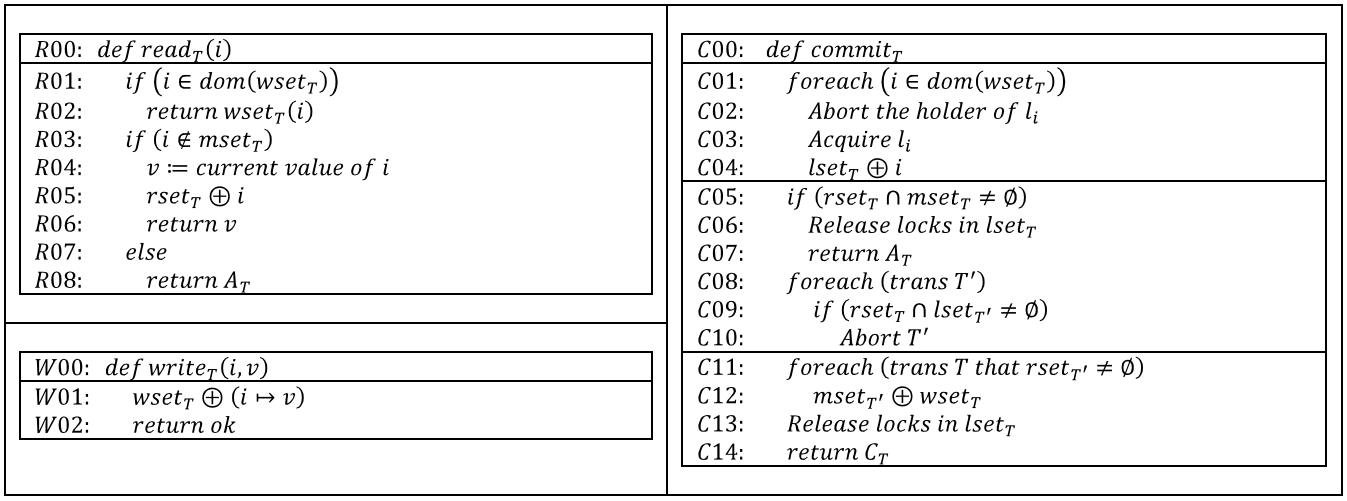


Figure 10.14: Specification of TL2

10.6 Related Works

We rewrite the transition systems of [31] and [24] to make them more readable. Figure 10.13 presents the specification of DSTM algorithm according to [31] and [24]. Figure 10.14 presents the specification of TL2 algorithm according to [31]. The specification of TL2 in [24] is the same as Figure 10.14 except that there is no horizontal line between and and $C10$ and $C11$. Horizontal lines separate fragments that are assumed to be executed atomically.

Chapter 11

Bibliography

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, May 1995.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[4] Woongki Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. 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, 2010.

[5] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[7] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.

[8] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, April 2007.

[10] Alexandre Buisse, Lars Birkedal, and Kristian Støvring. Step-indexed kripke model of separation logic for storable locks. *Electron. Notes Theor. Comput. Sci.*, 276:121–143, September 2011.

[11] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.

[12] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, 2008.

[13] Joey W. Coleman and Cliff B. Jones. A structural proof of the soundness of rely/guarantee rules. *J. Log. and Comput.*, 17(4):807–841, August 2007.

[14] Intel Corporation. Intel architecture instruction set extensions programming reference. 319433-012, 2012.

[15] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[16] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC, (LNCS 4167)*, 2006.

[19] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *SPAA*, 2010.

[20] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[23] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI*, 2009.

[24] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, June 2010.

[25] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[26] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 173–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, pages 19–37, Berlin, Heidelberg, 2007. Springer-Verlag.

[28] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[29] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *ACM SIGPLAN Conference on Programming Languages Design and Implemen tation*, pages 372–382, 2008.

[30] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of CAV'09, Seventh International Conference on Computer Aided Verification*, pages 321–336, 2009.

[31] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Model checking transactional memories. *Distributed Computing*, 2010.

[32] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[33] R. Haring, M. Ohnmacht, T. Fox, M. Gschwind, D. Sattereld, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, , and C. Kim. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, 2012.

[34] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, second edition, 2010.

[35] Tim Harris, Simon Marlow, Simon Peyton, and Jones Maurice Herlihy. Composable memory transactions. In *PPOPP'05*, pages 48–60. ACM Press, 2005.

[36] Jonathan Hayman and Glynn Winskel. Independence and concurrent separation logic. In *In Proc. LICS 06*. IEEE Press, 2006.

[37] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 440–451, London, UK, UK, 1998. Springer-Verlag.

[38] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[39] M. Herlihy, V. Luchangco, M. Moir, and III W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[40] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[41] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[42] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[43] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 353–367, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 280–281, New York, NY, USA, 2009. ACM.

[45] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 14–26, New York, NY, USA, 2001. ACM.

[46] Cliff B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83*, volume 9 of *IFIP Congress Series*, pages 321–332, Paris, France, September 1983. IFIP, North-Holland
slashIFIP.

[47] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.

[48] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[49] Mohsen Lesani. On the correctness of transactional memory algorithms, the companion. `http://www.cs.ucla.edu/~lesani/companion/dissertation`.

[50] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *CONCUR*, 2012.

[51] Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place. In *WTTM'12*, 2012.

[52] João Lourenço and Gonçalo Cunha. Testing patterns for software transactional memory engines. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, pages 36–42, New York, NY, USA, 2007. ACM.

[53] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 134–143, New York, NY, USA, 2006. ACM.

[54] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[55] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[56] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 69–80, New York, NY, USA, 2007. ACM.

[57] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–426, July 1981.

[58] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62, 2008.

[59] Leonor Prensa Nieto. The rely-guarantee method in isabelle/hol. In *Proceedings of the 12th European conference on Programming*, ESOP'03, pages 348–362, Berlin, Heidelberg, 2003. Springer-Verlag.

[60] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[61] Peter W. OHearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 268–280, New York, NY, USA, 2004. ACM.

[62] J. O'Leary, B. Saha, and Mark R. Tuttle. Model checking transactional memory with spin. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*, pages 335–342, 2009.

[63] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, December 1976.

[64] S. Owre, J.M. Rushby, and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated DeductionCADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg, 1992.

[65] V. Pankratius, A.-R. Adl-Tabatabai, , and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical Report 2009–12, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, September 2009.

[66] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. *SIGPLAN Not.*, 40(1):247–258, January 2005.

[67] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[68] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Notices*, 45(5), January 2010.

[69] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[70] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT*, 2006.

[71] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.

[72] Vasu Singh. *Formalizing and Verifying Transactional Memories*. PhD thesis, Ecole Polytechnique Federale de Lausanne, April 2010.

[73] Vasu Singh. Runtime verification for software transactional memories. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 421–435, Berlin, Heidelberg, 2010. Springer-Verlag.

[74] Serdar Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, apr 2008.

[75] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011.

[76] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Lus Caires and VascoT. Vasconcelos, editors, *CONCUR 2007 Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2007.