Transaction Protocol Verification with Labeled Synchronization Logic

Mohsen Lesani
Computer Science and Engineering Department
University of California, Riverside
lesani@cs.ucr.edu

Abstract

Synchronization algorithms that provide the transaction interface are intricate. We present an algorithm description language that explicitly captures the type of the used synchronization objects and associates labels to method calls to explicitly capture their intra-thread order. We use the language to capture architecture independent representations of transactional memory algorithms. We present a novel logic that enables reasoning about synchronization algorithms that are described in the language. The logic quantifies over program labels and provides specific predicates and intuitive inference rules to reason about the inter-thread execution and linearization orders of labeled method calls. In particular, the logic assertions can directly capture orders that are fundamental to the correctness of transactions. We present a denotational semantics for the language and prove the soundness of the logic. We have formalized the logic in the PVS proof assistant and mechanically constructed the challenging correctness proof of the TL2 transactional memory algorithm.

1 Introduction

Synchronization algorithms such as mutual exclusion, concurrent data structures and transactional memory are subtle. Designing a synchronization algorithm involves choosing *synchronization objects* and programming the coordination logic using them. There is a trade-off in the choice between consistency and efficiency of a synchronization object. For example, although an atomic register maintains consistency in every concurrent execution, it is less efficient than a basic register that does not provide any guarantee in the presence of race. In addition, algorithm designers have to decide and properly program the *order* of method calls in each thread. These orders are crucial to the correctness of the algorithm in every possible interleaving. Intra-thread orders are usually specified using architecture-dependent fence instructions. As a result, a synchronization algorithm is complicated, low-level and prone to bugs. Engineering reliable software stacks built on top of these algorithms requires their precise description and rigorous verification. In this paper, we present a *description language* for synchronization algorithms, a novel *logic* to reason about synchronization algorithm descriptions and apply the logic to *mechanize* the verification of transactional (TM) memory algorithms.

Dekker Example in the Description Language. The language explicitly captures the type of the base synchronization objects and the intra-thread order of method calls on them. As an example, consider the description of the Dekker mutual exclusion algorithm in Figure 1.(a). The description comprises three sections. The first section, typing $\mathcal{T}$, describes the base synchronization objects that the algorithm uses. Dekker uses two atomic registers as flags. Using basic registers as flags can lead to a race and violation of mutual exclusion. The second section, definitions $\mathcal{D}$, describes the definition of methods. The initialization method initializes the two flags to zero. In the two try-lock methods, each thread first writes to its own flag and then reads from the flag of the other thread. Each try-lock method allows entry to the critical region only if it finds the flag of the other thread unset. Every method call in the description is uniquely marked with a *label*. 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 method calls can violate mutual exclusion. The required orders for the body of each method definition are declared after the body. For example, the order $W_1 \rightarrow R_2$ requires the methods call W_1 to be executed before the method call R_2. The third section, program $\mathcal{P}$, represents the concurrent client program for the defined methods. First, the initialization method is called and then two concurrent threads are executed, each calling one of the try-lock methods.

Reasoning and Orders. The mutual exclusion property states that at most one of the two threads can enter the critical section. More precisely, if either of the two methods calls labeled L_1 and L_2 returns *true*, the other one returns *false*. An intuitive classical proof for the mutual exclusion property of the Dekker algorithm is as follows. We directly reason about execution and linearization orders across threads and use the properties of linearizable registers such as the totality of the linearization order. In particular, we use the real-time-preservation property [19] of linearizability that states that if a method call is executed before another on a linearizable object, then the former is linearized before the latter as well. Assume that L_1 returns true, we prove that L_2 does not return true. If L_1 has returned true, it should have been by C_{1t}.

NFM'19, May 2019,
2019.

$\mathcal{T}$: f_1, f_2 : **AtomicRegister**		
$\mathcal{D}$:		
def *init*()	**def** $tryLock_1()$	**def** $tryLock_2()$
$W_{01} \triangleright$ $f_1.write(0)$,	$W_1 \triangleright$ $f_1.write(1)$,	$W_2 \triangleright$ $f_2.write(1)$,
$W_{02} \triangleright$ $f_2.write(0)$,	$R_2 \triangleright$ $x_2 = f_2.read()$,	$R_1 \triangleright$ $x_1 = f_1.read()$,
	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*(),		
$L_1 \triangleright$ $l_1 = tryLock_1()$ $\parallel$ $L_2 \triangleright$ $l_2 = tryLock_2()$		

(a)

X2L

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

P2X

$$\frac{c_1 \rightarrow_\pi c_2 \qquad \pi,\Gamma \vdash exec(\varsigma' c_1) \qquad \pi,\Gamma \vdash exec(\varsigma' c_2)}{\pi,\Gamma \vdash \varsigma' c_1 \prec \varsigma' c_2}$$

(b)

Figure 1. (a) Dekker Algorithm Description $\pi_D = (\mathcal{T}, \mathcal{D}, \mathcal{P})$. (b) Example inference rules.

Therefore, the condition of the if statement is satisfied (that is $x_2 = 0$). Therefore, the read operation R_2 from the flag f_2 returns 0. There are two write method calls W_{02} and W_2 on f_2. The initialization method call W_{02} is executed before both R_2 and W_2; therefore, by the real-time preservation property, is linearized before them. Now, W_2 can either linearize before or after R_2. The first case is not possible because otherwise, W_2 would be the last write to f_2 before R_2. Therefore, R_2 would return the value that W_2 writes. However, W_2 writes 1, and R_2 has returned 0.

In the second case, the method call R_2 linearizes before W_2. Therefore, (1) R_2 is executed before or concurrent to W_2. (This holds because otherwise, R_2 would be executed after W_2. Thus, by the real-time-preservation property, R_2 would be linearized before W_2 as well that contradicts the assumption of this case.) According to the explicit program orders, (2) W_1 is executed before R_2 and (3) W_2 is executed before R_1. From the transitivity of the execution order on the three orders 2, 1, and 3 above, we have that W_1 is executed before R_1. Therefore, W_1 linearizes before R_1 as well. The initialization method call W_{01} is executed before R_1 and W_1 and is therefore linearized before the two. Therefore, W_1 is the last write to f_1 before R_1. Therefore, R_1 returns the value that W_1 writes that is 1. Therefore, $x_1 = 1$. Therefore, as the condition of the if statement is not satisfied, C_{2f} is executed. Therefore, L_2 returns false.

Labeled Synchronization Logic. Can we build a rigorous foundation for this intuitive style of reasoning? Is it possible to construct formal proofs in this style? We present a novel first-order logic called *labeled synchronization logic (LSL)* that enables reasoning about synchronization algorithms based on the *execution and linearization orders* of method calls on the base synchronization objects. It quantifies over program labels and provides specific predicates for execution order, execution overlap and linearization orders of labeled method calls across threads. These assertions capture critical orders between concurrent operations. In addition, LSL provides simple-to-use *inference rules* to reason about these orders and deduce algorithm correctness. For example, we applied LSL to state and prove the mutual exclusion property of the Dekker algorithm. The following theorem states that for the Dekker description (π_D) with no prior assumptions ($\cdot$), it can be inferred that if L_1 returns true, then L_2 returns false. If the first thread can enter the critical section, then the second cannot. The symmetric property can be state and proved similarly.

Theorem 1.1.
$\pi_D, \cdot \vdash (retv(L_1) = true) \Rightarrow (retv(L_2) = false)$.

The full proof is available in the appendix [28] § 12 and the mechanised proof is available at [27]. We show two inference rules of LSL as examples in Figure 1.(b). LSL uses dynamic labels l to uniquely identify method call instances. For example, the label L_2'W_2 is a call string that refers to the instance of the method call labeled W_2 (in the definitions section $\mathcal{D}$) that is executed in the body of the caller method labeled L_2 (in the program section $\mathcal{P}$).

The rule X2L states the real-time-preservation property of linearizability [19]: the execution order $\prec$ of method calls on an object o of a linearizable type LT is preserved in the linearization order $\prec_o$ of the object o. If a method call is executed before another on a linearizable object, then the former is linearized before the latter as well. Using the rule X2L, a step that we saw in the informal proof of the Dekker algorithm can be formalized as follows. From the fact that method call L_0'W_{02} is executed before $\prec$ the method call L_2'W_2 and that both are on the atomic register f_2,

$$\pi_D, \Gamma \vdash obj(L_0\text{'}W_{02}) = obj(L_2\text{'}W_2) = f_2 \ \wedge \ L_0\text{'}W_{02} \prec L_2\text{'}W_2$$

we can deduce that L_0'W_{02} is before L_2'W_2 in the linearization order $\prec_{f_2}$ of the atomic register f_2

$$\pi_D, \Gamma \vdash L_0\text{'}W_{02} \prec_{f_2} L_2\text{'}W_2$$

In the sequent, Γ is any set of assumption assertions.

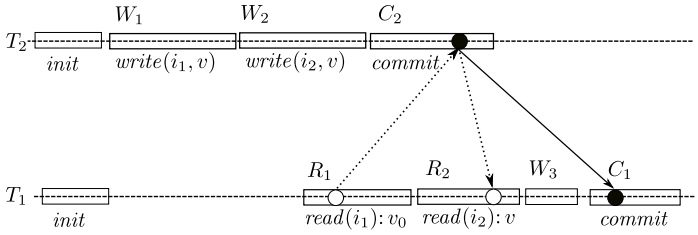

Figure 2. Illustration of Read-Preservation

The the rule P2X states the *program-order-preservation* property: the program order is preserved in the execution order. (The prefix label variable is denoted by ς.) For example, the Dekker description π_D declares the method call W_1 to be ordered before the method call R_2 i.e. $W_1 \rightarrow_\pi R_2$. Using the rule P2X, from the declared order and that both L_1'W_1 and L_1'R_2 are executed

$$\pi_D, \Gamma \vdash exec(L_1\text{'}W_1) \ \wedge \ exec(L_1\text{'}R_2)$$

we can deduce that L_1'W_1 is executed before L_1'R_2

$$\pi_D, \Gamma \vdash L_1\text{'}W_1 \prec L_1\text{'}R_2$$

Correctness of Transactional Memory. Execution and linearization orders of method calls on the base synchronization objects play a critical role in the reasoning about the correctness of transactional memory algorithms. In a previous work [31], we represented a decomposition called *markability* of the TM correctness condition opacity [17]. Markability decomposes opacity to separate intuitive invariants that can be separately verified. An execution history is markable if there is a specific ordering relation on the set of transactions and their read operations called marking such that three invariants are satisfied.

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 orders: The *effect order*: The effect order is a total order of the transactions. It represents the order in which the transactions appear to take effect. The *access orders*: Let us refer to the committed transactions that have write operation(s) to location i as *writers of* i. Consider an unaborted read operation R on a location i. For each such R, the access order is an antisymmetric relation that orders R and every writer of i. The access order of R represents where R has read i between the writers of i.

For example, Figure 2 presents the sketch of a transaction history with two transactions T_1 and T_2. The horizontal lines from left to right show the time for the two threads executing T_1 and T_2, and the boxes show execution of method calls on the transactional interface. These method calls may call multiple methods on the base synchronization objects. The dark circles show the effect points of the two transactions and the solid arrow shows the effect order. The transaction T_2 takes effect before T_1. The transaction T_2 is a writer of both i_1 and i_2. The white circles show the access points for the two reads R_1 and R_2. The read R_1 reads i_1 before T_2 writes to it. Similarly, the read R_2 reads i_2 after T_2 writes to it. The relation is called marking as these points can be usually marked as particular *method calls on the base synchronization objects* in the algorithm and the orders are defined as execution and linearization orders on these method calls.

As an example, the second invariant of markability called *read-preservation* requires that the location read by a read operation is not overwritten between the two points that the read takes place and the transaction takes effect. Consider an unaborted read operation R from a location i by a transaction T. Intuitively, read-preservation requires that no writer of i comes between R and T in the marking relation. The read-preservation property is violated in Figure 2. The read R_1 is an unaborted read from i_1 in T_1. The transaction T_2 is a writer of i_1. The read R_1 is before T_2 in the access order and T_2 is before T_1 in the effect order. The value that R_1 reads is overwritten by T_2 before T_1 is committed. The transaction T_2 writes a new value to both i_1 and i_2. The read R_1 reads the old value of i_1 and R_2 reads the new value of i_2 that can be inconsistent. Read-preservation is usually simply verified by the validation checks in the commit and read operations.

In this project, We have used LSL to construct a new mechanized proof of TL2 [10], in the PVS proof assistant. Specifically, we have *expressed markability* in the assertion language and then applied LSL inference rules to deduce the markability assertion. This result shows that LSL is scalable to complicated transactional memory algorithms.

Semantics and Soundness. We have proved the *soundness* of LSL: If an assertion is deduced using valid assumptions, then the deduced assertion is valid as well. An assertion is valid if it evaluates to true in every execution. To prove the soundness of LSL, we defined a novel denotational semantics of the description language. The semantics has the following desirable properties: (1) *Compositional* for base objects. The semantics is defined abstract from specific object types and can be modularly augmented with new types. The semantics of objects are separately defined. (2) *True concurrency*. In the interleaving semantics for concurrency, whole method calls are the elements of execution histories. In contrast, true concurrency is more fine-grained and considers invocation and response events of method calls. Therefore, the semantics not only supports atomic but also non-atomic objects. (3) *Relaxed execution.* [4, 21] Methods that are not required to be ordered by the description are allowed to execute out-of-order. Thus, the semantics models the behavior of common architectures that reorder instructions of a thread.

The Structure of the Paper. In § 2 and 3, we present the description language and LSL, and in § 4, we present the semantics of the language and prove the soundness of LSL. In § 5, we state a marking for TL2 and prove the markability of TL2 in LSL. We present the related works in § 6 before conclusion.

$$
\begin{array}{rcl}
\pi \in \Pi & ::= & (\mathcal{T}, \mathcal{D}, \mathcal{P}) \\
\mathcal{T} & ::= & (\phi : ot)^* \\
\mathcal{D} & ::= & d^* \\
d & ::= & \textbf{def}\ n_t(x^*)\ s, r \\
s & ::= & s, s \mid \textbf{if}\ (b)\ s\ \textbf{else}\ s \mid q \\
 & \mid & c \triangleright\ \textbf{foreach}\ (i \in set)\ s \\
q & ::= & c \triangleright\ x = o.n_\tau(u^*) \\
 & \mid & c \triangleright\ \textbf{return}\ u \\
b & ::= & \neg b \mid b\ \wedge\ b \mid u = u \mid u = u + u \mid u < u \\
r & ::= & \text{`\{'}\ (c \rightarrow c)^*\ \text{`\}'} \\
\mathcal{P} & ::= & p_0, (p_1 \| p_2 \| \ldots \| p_n) \\
p & ::= & p; p \mid \textbf{if}\ (b)\ p\ \textbf{else}\ p \mid c \triangleright\ x = n_\tau(u^*)
\end{array}
$$

Figure 3. Algorithm Description Grammar.

2 Description Language

We now present the language that we describe concurrent algorithms in. An algorithm description π is a triple $(\mathcal{T}, \mathcal{D}, \mathcal{P})$ where $\mathcal{T}$ is the *type declarations* for the used synchronization objects, $\mathcal{D}$ is *the method definitions*, and $\mathcal{P}$ is a concurrent *client program* that calls the defined methods. The set of descriptions Π is defined in Figure 3. The description of the Dekker algorithm is presented in Figure 1.(a) as an example. We look at each section in turn.

A typing $\mathcal{T}$ is a mapping from object names ϕ to object types ot. We use x^* to denote a finite sequence of x's. An object type ot is either a scalar or an array type. A scalar type is either a basic type BT such as BasicRegister, BasicSet, or BasicMap, or a linearizable type LT such as AtomicRegister, AtomicCASRegister, Lock, TryLock, or strong counter SCounter. As an example, in the Dekker description of Figure 1.(a), both flags f_1 and f_2 are declared to be atomic registers. Using basic registers can lead to a race and violation of mutual exclusion. We will revisit synchronization object types when we present their specific inference rules. An array type of a scalar type st is of the form $st[\,]$. A thread-local type is an array type and the well-formedness conditions enforce that a thread-local object is only indexed by the identifier of the calling thread.

The definitions $\mathcal{D}$ is a sequence of method definitions d. We denote a method name by n, a value by v, a variable by x, a value or variable by u, a thread value by T, a thread variable by t, and a thread value or variable by τ. The method definition $\textbf{def}\ n_t(x^*)\ s, r$ defines a method named n with thread parameter t and data parameters x^* with the body s and the declared order r. The Dekker description of Figure 1.(a) defines three methods: $init$, $tryLock_1$ and $tryLock_2$. A statement s is either a sequence, a conditional, a method call or a return statement. 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, x is the return variable, o is the receiving object, n is the method name, τ is the current thread argument, and u^* are the data arguments. The labels of statements are unique in π. Every variable is uniquely bound. An object o is either a single object ϕ or an an element of an array $\phi[u]$. In a return statement $c \triangleright\ \textbf{return}\ u$, c is the label and u is the returned value or variable. In the appendix [28] § 9, we define the **foreach** iteration statement on sets and maps as a syntactic sugar. As we will see, the semantics of the language supports out-of-order or *relaxed* execution. Any two labels that are left unordered by the description may be reordered in the execution. Data and control dependencies in the method body s impose order between statements. However, the programmer can explicitly require additional orders. The declared program order r of a method definition is a binary relation on the set of labels in the body s. For example, the Dekker description of Figure 1.(a) declares the orders $W_1 \rightarrow R_2$ and $W_2 \rightarrow R_1$ that are crucial to the correctness. These orders are usually enforced by fence instructions. Programming fences is complicated and error-prone. The platform-independent description of the required orders can be used by compilers to optimize fence insertion [5] for different target architectures. Further, the declared order facilitates architecture-independent verification. In addition, if the order of two statements that is unnecessary for correctness is changed, the proof stays unchanged.

The client program section $\mathcal{P}$ is of the form $p_0, (p_1 \| p_2 \| \ldots \| p_n)$ where p_0 is the initialization program, and p_1, p_2, ..., and p_n are the parallel programs. For example, the program section of the Dekker description in Figure 1.(a) has two parallel programs that each call one of the two defined try-lock methods. A sequential program p is either a sequence, a conditional or a method call. In a method call $c \triangleright\ x = n_\tau(u^*)$, n is name of a method that is defined in the method definitions section $\mathcal{D}$. The object **this** is the default receiver object and is therefore elided in the client calls. We use θ to denote a synchronization object o or the **this** object.

Let $\rightarrow_n$ denote the irreflexive transitive closure of the data and control dependencies and the declared order of method 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 the labels of p_0 before the labels of parallel programs), (2) the sequential order of the sequential programs p_i, and (3) for each method definition n, the order $\rightarrow_n$.

LSL uses the following functions that are directly derived from the program description. The names of methods defined in a description are unique. Thus, we define the functions $par1_\pi$ and $par2_\pi$ that map method names to their first and second parameters. Similarly, $tpar_\pi$ maps method names to their thread parameter. As the labels in a description are unique, we define the function obj_π that maps the label of a method call to its receiver object. Similarly, the functions $index_\pi$, $name_\pi$, $thread_\pi$, $arg1_\pi$, $arg2_\pi$ and $retv_\pi$ map the label of a call to the array index of the receiver object, the name of the method, the thread identifier, first and second arguments and the return variable of the method call. For a return

statement, we let $name_\pi$ and $arg1_\pi$ map to the name *return* and the argument of the return statement respectively.

We call the conjunction of all the enclosing if (and else) conditions of a statement, its *enclosing condition*. Let the function $cond_\pi$ map statement labels to their enclosing conditions. Let $Labels_\pi(n)$ denote the set of labels in the body of the method n. Let $Returns_\pi(n)$ denote the set of labels of return statements in the body of n. Let $PreReturns_\pi(c)$ denote the set of labels of the return statements before the statement labeled c in π.

3 Labeled Synchronization Logic

Now, we present our first-order logic to reason about synchronization algorithm descriptions.

Assertions. We first define the set of dynamic labels that uniquely identify method call instances. As an example, in execution histories for the Dekker algorithm (Figure 1.(a)), we have the two labels L_1 and L_1'W_1. The label L_1 refers to a call site in the program section $\mathcal{P}$. On the other hand, the label L_1'W_1 is a call string that refers to the instance of the method call labeled W_1 (in the definitions section $\mathcal{D}$) that is executed in the body of the client method call labeled L_1. Thus, a dynamic label l is of the form ς'c where the pre-label ς is either a static label c or no label ϵ. The symbol ϵ is the left identity element of the prefixing operator.

A method can be called several times in the program. To have unique local variable names, every variable (including the parameters) of the method is prefixed by the caller label. The labeled variable c'x denotes the instance of the local variable x in the body of the method call labeled with c. Similarly, the prefixing operator is lifted to expressions over variables. The assertions language of LSL is described by the following grammar.

$$
\begin{array}{lcll}
e & ::= & obj(\ell) \mid name(\ell) \mid thread(\ell) \mid & \text{Expression} \\
 & & arg1(\ell) \mid arg2(\ell) \mid retv(\ell) \mid & \\
 & & initOf(\tau) \mid commitOf(\tau) \mid & \\
 & & o \mid n \mid \varsigma\text{'}x \mid v \mid \varsigma\text{'}t \mid T & \\
\mathcal{R} & ::= & e = e \mid e < e \mid \ell = \ell' \mid c = c \mid & \text{Proposition} \\
 & & exec(\ell) \mid \ell \prec \ell' \mid \ell \sim \ell' \mid \ell \prec_o \ell' \mid \tau \lessdot \tau' & \\
\mathcal{A} & ::= & \mathcal{R} \mid \neg\mathcal{A} \mid \mathcal{A} \wedge \mathcal{A} \mid \forall \ell : \mathcal{A} \mid \forall t : \mathcal{A} & \text{Assertion}
\end{array}
$$

Here, c is a program label (such as C_{1t} in the Dekker description of Figure 1.(a)), l is a constant label (such as the dynamic labels L_1 and L_1'C_{1t} for the Dekker description), ℓ is a label variable, T is a thread (or transaction) identifier value, t is a thread identifier variable, τ is a thread value or variable, x is a variable, v is a value, o is an object name, and n is a method name. Expressions use six function symbols. The functions *obj*, *name*, *thread*, *arg*1, *arg*2 and *retv* map a label of the program to its object, method name, thread name, first and second argument, and return value. The function *initOf* maps each transaction to the label of its *init* method call. The function *commitOf* maps each committed transaction to the label of its *commit* method call. Propositions use seven predicates. The first two are equality (=) and integer comparison (<). The proposition $exec(\ell)$ states that the method call labeled ℓ is *executed*. The proposition $\ell \prec \ell'$ asserts that ℓ is *executed before* ℓ'. the proposition $\ell \sim \ell'$ asserts that ℓ is *executed concurrent* to ℓ'. For a linearizable object o, the proposition $\ell_1 \prec_o \ell_2$ states that ℓ_1 is *linearized before* ℓ_2 in the linearization order of o. (As we will describe in the semantics section, any concurrent execution on a linearizable object is equivalent to a correct sequential execution. The total order of method calls in that sequential execution is called the linearization order.) The proposition $\tau \lessdot \tau'$ asserts that all the labels of thread τ are executed before all the labels of thread τ'. This assertion is used to state that a transaction is executed before another.

An assertion is either a proposition, negation of an assertion, conjunction of two assertions, or existential quantification over labels or transactions. As usual, we can define, disjunction $\vee$, universal quantification $\forall$, less than or equal $\leq$, executes before or equal $\preceq$, linearized before or equal $\preceq_o$, thread executed before or equal $\lessdot$ as syntactic sugar.

For an algorithm description π, a judgement is of the form $\pi, \Gamma \vdash \mathcal{A}$, where Γ is the context, that is, a list of assertions, and $\mathcal{A}$ is a closed assertion. The judgement is read as for every execution of the program π, if the assertions in Γ hold, then the assertion $\mathcal{A}$ holds. For example, $\pi, \Gamma \vdash l \prec l'$ says that in every execution of π where Γ holds, the statement labeled l is executed before the statement labeled l'.

Algorithms can be verified modularly. The client program section $\mathcal{P}$ of an algorithm description π can specify general clients. For example a lock algorithm description π (such as Dekker) can be verified separately. Then, the lock object (with its abstracted implementation) can be used to implement a TM. Verification of the description π' of the TM is restricted to the labels of π' and does not involve π.

Inference Rules. We now present the inference rules of LSL. The inference rules can be conceptually divided into four groups. First, the first-order logic rules (which are standard and omitted here). Second, the basic rules that axiomatize the properties of execution and linearization orders and their interdependence (Figure 4). Third, the synchronization object rules that axiomatize the properties of common synchronization object types (Figure 5). Fourth, the inference rules that axiomatize the relation of the algorithm description and the execution (Figure 6). We showcase a few rules. The full set of rules for the common synchronization objects is available in the appendix [28] § 11.

Figure 4 represents the set of basic inference rules that intuitively capture the properties of execution and linearization orders and their relation. 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 execution order. The rule X2Trans states the

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

X2Trans
$$\frac{\pi,\Gamma \vdash l_1 \prec l_2 \quad \pi,\Gamma \vdash l_2 \sim l_3 \quad \pi,\Gamma \vdash l_3 \prec l_4}{\pi,\Gamma \vdash l_1 \prec l_4}$$

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

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

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

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

L2X
$$\frac{\pi,\Gamma \vdash l \prec_o l'}{\pi,\Gamma \vdash exec(l) \ \wedge \ exec(l') \ \wedge \ obj(l) = obj(l') = o}$$
(c)

Figure 4. The Basic Inference Rules. (a) properties of execution orders, (b) linearization orders, and (c) derived rules.

AReg
$$\frac{\mathcal{T}(r) = AtomicRegister \qquad \pi,\Gamma \vdash isRead_r(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isWriter_r(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

BReg
$$\frac{\mathcal{T}(r) = BasicRegister \qquad \pi,\Gamma \vdash isSingleWriter(r) \qquad \pi,\Gamma \vdash isRead_r(l_R) \qquad \pi,\Gamma \vdash isRaceFree_r(l_R)}{\pi,\Gamma \vdash \exists \ell_W : isEWriter_r(\ell_W, l_R) \wedge retv(l_R) = arg1(\ell_W)}$$

LockUnlockPair
$$\frac{\mathcal{T}(o) = Lock \qquad \pi,\Gamma \vdash isOwnerRespect(o) \qquad \pi,\Gamma \vdash isLock_o(l_{a_1}) \qquad \pi,\Gamma \vdash isUnlock_o(l_{r_2}) \qquad \pi,\Gamma \vdash l_{a_1} \prec_o l_{r_2}}{\pi,\Gamma \vdash \exists \ell_{r_1}, \ell_{a_2} : isUnlock_o(\ell_{r_1}) \ \wedge \ thread(\ell_{r_1}) = thread(l_{a_1}) \wedge isLock_o(\ell_{a_2}) \ \wedge \ thread(\ell_{a_2}) = thread(l_{r_2}) \wedge \ell_{r_1} \prec_o \ell_{a_2}}$$

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

$isRead_r(\ell_R) \Leftrightarrow$
$\quad exec(\ell_R) \ \wedge \ obj(\ell_R) = r \ \wedge \ name(\ell_R) = read$
$isWrite_r(\ell_W) \Leftrightarrow$
$\quad exec(\ell_W) \ \wedge \ obj(\ell_W) = r \ \wedge \ name(\ell_W) = write$
$isWriter_r(\ell_W, \ell_R) \Leftrightarrow$
$\quad isWrite_r(\ell_W) \ \wedge \ \ell_W \prec_r \ell_R \wedge$
$\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq_r \ell_W \ \vee \ \ell_R \prec_r \ell'_W)$
$isEWriter_r(\ell_W, \ell_R) \Leftrightarrow$
$\quad isWrite_r(\ell_W) \ \wedge \ \ell_W \prec \ell_R \wedge$
$\quad \forall \ell'_W : isWrite_r(\ell'_W) \Rightarrow (\ell'_W \preceq \ell_W \ \vee \ \ell_R \prec \ell'_W)$
$isRaceFree_r(\ell) \Leftrightarrow$
$\quad \forall \ell_W : isWrite_r(\ell_W) \Rightarrow (\ell_W \prec \ell \ \vee \ \ell \prec \ell_W)$
$isSingleWriter(r) \Leftrightarrow$
$\quad \forall \ell_w : isWrite_r(\ell_w) \Rightarrow isRaceFree_r(\ell_w)$

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

Figure 5. Synchronization Object Inference Rules. Four of the rules for atomic and basic registers, lock and strong counter.

transitivity of the sequence of precedence, concurrency and precedence execution relations. If l_1 is executed before l_2, l_2 is executed 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 pair of executed method calls either execute in order or concurrently. The rule X2L states the *real-time-preservation* property of linearization orders: The execution order of two method calls on a linearizable object (specified by $\mathcal{T}(o) \in LT$) 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. Every two executed method calls on

$$\textsc{P2X}\quad \frac{c_1 \rightarrow_\pi c_2 \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_1) \qquad \pi,\Gamma \vdash exec(\varsigma\text{'}c_2)}{\pi,\Gamma \vdash \varsigma\text{'}c_1 \prec \varsigma\text{'}c_2}$$

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

Figure 6. Inference Rules about the Algorithm Description.

a linearizable object are ordered in its linearization order. The two derived rules X2X and L2X can be established by an inductive reasoning on the length of the proof of $l \prec_o l'$. The rule X2X states that if a method call is executed before another one, then clearly both are executed. The rule L2X states that if a method call is linearized before another one, then clearly both are executed.

Now let us look at a few synchronization object rules in Figure 5. First, 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 reads and ℓ_W is the last write method call that is linearized before l_R. Second, the rule BReg states that if a basic register r is single-writer, for every race-free read method call l_R on r, there is a write method call ℓ_W on r that writes the same value that l_R reads and ℓ_W is the last write method call that is executed before l_R. A register r is single-writer 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. A method call r is race-free if and only if there is no *write* method call on r that executes concurrent to it. The rules AReg and BReg model Lamport's notion of atomic and safe registers [26]. Third, the rule LockUnlock-Pair states the *lock-unlock-pair* property: if ownership of a lock object o is respected and a *lock* method call on o by a thread τ_1 is linearized before an *unlock* method call on o by a thread τ_2, then an *unlock* method call on o by τ_1 is linearized before a *lock* method call on o by τ_2. The rule is derived from the fact that if the ownership of a lock is respected, its linearization order is a sequence of matching pairs of *lock* and *unlock* method calls. Intuitively, ownership for a lock o is respected, if and only if every thread unlocks o only if it has already locked o and has not unlocked o since then. Fourth, the rule CountSeq states the *count-sequence* property: for a strong counter object co, if the return value of an *iaf* (inc-and-fetch) method call on co is less than the return value of another method call on co, then the former is linearized before the latter. The rule is derived from the fact that the return values of method calls in the linearization order of a strong counter is non-decreasing.

The rules presented in Figure 6 refer to the algorithm description. The rule P2X states the *program-order-preservation* property: the program order is preserved in the execution order. If the algorithm description requires a method call c_1 to be ordered before another method call c_2, the order is preserved in the execution of them from any call site ς. That is if ς'c_1 and ς'c_2 are executed, then ς'c_1 is executed before ς'c_2. The rule Callee states that if a method call c' in the body of the caller method call ς is executed, then the later is also executed, is a *this* method call and its parameters and arguments are equal.

4 Semantics and Soundness

Now, we first present execution histories and semantics of objects and then, present the semantics of the language. The language semantics is used to state the soundness of LSL.

Strings. Let s be an isogram string (i.e. contains no repeating occurrence of the alphabet.) For any s_1, s_2, 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.

Method calls and events. An *invocation event* is of the form $inv(l \triangleright\ o.n_T(v))$ where l is a label, o is an object, n is a method name, T is a transaction identifier and v is a value. A *response event* is of the form $ret(l \triangleright\ v)$ where l is a label and v is a value. A *completed* method call is the sequence of an invocation event and the matching response event (with the same label). We use $l \triangleright\ o.n_T(v)\text{:}v'$ 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*. We use $E \cdot E'$ to denote the *concatenation* of E and E'. For a thread T, we use $E|T$ to denote the subsequence of all events by T in E. For an object o, we use $E|o$ to denote the subsequence of all events on o in E. We write $E \equiv E'$ iff for every thread T, $E|T = E'|T$, and say that E and E' are equivalent or indistinguishable. A sequence of events is *sequential* if and only if it is a sequence of completed method calls possibly followed by an invocation event. Let $Sequential$ be the set of sequential sequences. A thread T is *sequential* in a sequence of events E if $E|T$ is sequential.

Execution history. An *execution history* is an event sequence where invocation events have unique labels and every transaction is sequential. We say label l is in X and write $l \in X$ if there is an invocation event with label l in X. We use l, R and W to denote labels. Let $Labels(X)$ denote the set of labels in X. An invocation event is pending in a history X if it has no matching response event in X. A history is complete if and only if it has no pending invocation event. As the labels are unique in a history, the following functions are defined. The functions obj_X, $name_X$, $trans_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the transaction identifier, the first and the second arguments, and the return value associated with the labels. Similarly, iEv and rEv functions map labels to the invocation and the response events associated with the labels.

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

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 $T \lessdot_X T' \ \vee \ T = T'$.

Abstract object types. An *abstract object* type o is represented by a method interface and a sequential specification. The *sequential specification* of an object o, denoted by $SeqSpec(o)$, is the prefix-closed set of correct sequential histories of o. We describe the register and lock abstract object types and leave other objects and their formal definitions to the appendix [28] § 10.2.

A register r is an object that encapsulates a value and supports *read* and *write* methods. The method call $r.read()$ returns the current encapsulated value of r. The method call $r.write(v)$ overwrites the encapsulated value of r with v. The sequential specification of r is the set of sequential histories of *read* and *write* method calls on r where every read returns the argument of the latest preceding write (regardless of thread identifiers). (It is assumed that a write method call initializes the register before other methods are invoked.) A lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: The method calls $l.lock()$ and $l.unlock$ change the state from $\mathbb{R}$ to $\mathbb{A}$ and $\mathbb{A}$ to $\mathbb{R}$ respectively. The method call $l.read()$ returns *true* if the state of l 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. 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 subhistory 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.

Concurrent object types. We consider two *concurrent object* types: basic and linearizable (atomic). Linearizable objects comply with their sequential specification in every concurrent execution. Basic objects comply with their sequential specification only if they are accessed sequentially.

An object is *basic* iff every sequential execution on it results in a history in its sequential specification. The semantics of a basic object o, is a set of histories $\mathbb{H}_B(o)$ such that $\mathbb{H}_B(o) \cap Sequential \subseteq SeqSpec(o)$. Note that the semantics of a basic object may contain arbitrary non-sequential histories. For example, a read from a basic register can return an arbitrary value in a race, i.e. when it is concurrent to a write.

An object o is *linearizable* iff for every complete execution history X on o, there is an indistinguishable sequential history L that is in the sequential specification of o and is real-time-preserving. The semantics of a linearizable object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and linearization pairs $\mathbb{H}_L(o) = \{(X,L) \mid X \equiv L \ \wedge \ L \in SeqSpec(o) \ \wedge \ \prec_X \subseteq \prec_L\}$. The history L is the linearization and $\prec_L$ is the linearization order of X. This definition is lifted to incomplete histories X by appending arbitrary response events for pending invocations.

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

$$[\![c \triangleright\ x' = n_\tau(u)]\!] = \{(X,\sigma) \mid \quad (1)$$

$$X = inv(c \triangleright\ n_\tau(u)) \cdot X' \cdot ret(c \triangleright\ x') \wedge \quad (2)$$

$$X' \in Sequential \wedge \quad (3)$$

$$\sigma(c'tpar_\pi(n)) = \sigma(\tau) \ \wedge \ \sigma(c'par1_\pi(n)) = \sigma(u) \wedge \quad (4)$$

$$Labels(X') \subseteq \{\overline{c'c_i}\} \wedge \quad (5)$$

$$\forall c'c_i \in X': \quad (6)$$

$$obj_{X'}(c'c_i) = c'obj_\pi(c_i) \ \wedge \ thread_{X'}(c'c_i) = c'thread_\pi(c_i) \wedge$$

$$name_{X'}(c'c_i) = name_\pi(c_i) \ \wedge \ arg1_{X'}(c'c_i) = c'arg1_\pi(c_i) \wedge$$

$$ret v_{X'}(c'c_i) = c'ret v_\pi(c_i) \wedge$$

$$\forall c_i \in \{\overline{c_i}\}: c'c_i \in X' \Leftrightarrow \quad (7)$$

$$\left(\sigma(c'cond_\pi(c_i)) \ \wedge \ \forall c_j \in PreReturns_\pi(c_i) \Rightarrow \neg c'c_j \in X'\right) \wedge$$

$$\forall c_i, c_j \in \{\overline{c_i}\}: \left((c_i \rightarrow_n c_j) \ \wedge \ c'c_i \in X' \ \wedge \ c'c_j \in X'\right) \Rightarrow \quad (8)$$

$$c'c_i \prec_{X'} c'c_j \wedge$$

$$\exists c_r \in \{\overline{c_r}\}: c'c_r \in X' \ \wedge \quad (9)$$

$$\forall c_r \in \{\overline{c_r}\}: c'c_r \in X' \Rightarrow \sigma(x') = \sigma(c'arg1_\pi(c_r))\} \quad (10)$$

$$[\![p_1;\ p_2]\!] = \{(X,\sigma) \mid \exists X_1, X_2: \quad (11)$$

$$(X_1,\sigma) \in [\![p_1]\!] \ \wedge \ (X_2,\sigma) \in [\![p_2]\!] \ \wedge \ X = X_1 \cdot X_2\}$$

$$[\![\textbf{if } b\ p_1 \textbf{ else } p_2]\!] = \{(X,\sigma) \mid ((X,\sigma) \in [\![p_1]\!] \ \wedge \ \sigma(b)) \vee \quad (12)$$

$$((X,\sigma) \in [\![p_2]\!] \ \wedge \ \neg\sigma(b))\}$$

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

$$[\![\pi]\!] = \{$$

$$(X,\sigma,\mathcal{L}) \mid \exists X_0,..,X_n: \forall i \in \{0..n\}: (X_i,\sigma) \in [\![p_i]\!] \wedge \quad (13)$$

$$X = X_0 \cdot X' \ \wedge \ X' \in Interleave(X_1,...,X_n) \wedge \quad (14)$$

$$\forall o: \mathcal{T}(o) \in BT \Rightarrow \sigma(X)|o \in \mathbb{H}_B(o) \wedge \quad (15)$$

$$\forall o: \mathcal{T}(o) \in LT \Rightarrow (\sigma(X)|o, \mathcal{L}(o)) \in \mathbb{H}_L(o)\} \quad (16)$$

$$\mathbb{H}(\pi) = \{X' \mid \exists (X,\sigma,\mathcal{L}) \in [\![\pi]\!] \ \wedge \ X' = \sigma(X)\} \quad (17)$$

Figure 7. History Semantics $\mathbb{H}(\pi)$ of $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$

Language Semantics. Now, we present the denotational semantics $[\![\pi]\!]$ of a description π. The semantics is compositional for synchronization types, models true concurrency, and allows relaxed execution. It defines the set of histories of a description as a set of constraints enforcing the structure of the program and the semantics of the base objects.

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. Similar is the definition of $\sigma(b)$, $\sigma(o)$, $\sigma(u)$ and $\sigma(\tau)$. We use σ to represent the values of variables at the end of the execution.

Consider a description $\pi = (\mathcal{T}, \mathcal{D}, \mathcal{P})$. The history semantics of π, $\mathbb{H}(\pi)$, is defined in Figure 7. We illustrate each definition in turn. The semantics of a call in $\mathcal{P}$ to a method definition in $\mathcal{D}$ is defined in Eq. 1. Consider a method call $c \triangleright x' = n_\tau(u)$. Every execution history X of this method call starts with the invocation of n and finishes with a response from n (Eq. 2). The enclosed history X' is an execution history of the body of n. As every thread is sequential, the execution history X' of the body of n is a sequential execution history (Eq. 3). 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 above, τ and u are the thread argument and the first argument respectively. In every method call, the parameters are equal to the arguments. To have unique variables, the variables are prefixed by the caller label c. The function σ represents the mapping from variables to values at the end of the execution (Eq. 4). Let the set of labels of method n be $\{\overline{c_i}\}$. Obviously, an execution of the body involves only the labels that are in the body itself. To have unique labels, the body labels c_i are prefixed by the caller label c (Eq. 5). If a method call labeled c_i is executed in the body of a method call labeled c, every variable in c_i is prefixed by c in the execution history. Thus, the components of every dynamic label c'c_i are the components of the method call labeled as c_i in the program prefixed with c (Eq. 6). 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 (Eq. 7). The semantics models relaxed execution. Two method calls inside the body can be executed out-of-order if the description does not require them to be executed in order. However, the execution order preserves the order required by the description (Eq. 8). Every execution of the body executes a return statement (Eq. 9) and the argument of an executed return statement is equal to the return variable (Eq. 10).

Eq. 11 states 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. Eq. 12 states that the execution histories of the if-then-else statement is the union of the execution histories of the then statement when the condition is true and the execution histories of the else statement when the condition is false.

Eq. 13 to 17 define the semantics of an algorithm description $\pi = (\mathcal{P}, \mathcal{D}, \mathcal{T})$. The semantics enforces the semantics of the program $\mathcal{P}$ and the definitions $\mathcal{D}$ and also the semantics of the base objects in $\mathcal{T}$. The history of the initialization program p_0 precedes the history of the parallel programs. An execution history of the parallel composition of a set of programs is an interleaving of execution histories of those programs. The semantics models true concurrency as histories can be interleaved at the level of invocation and response events rather than method calls (Eq. 14). The semantics $[\![\pi]\!]$ represents an execution as a triple $\mathcal{X} = (X, \sigma, \mathcal{L})$. The history X is a *symbolic* execution history of π where variables are not yet instantiated to values. The mapping σ represents the values of variables at the end of the execution. The mapping $\mathcal{L}$ maps linearizable objects to their linearization histories. Applying σ to X yields a *concrete* execution history $\sigma(X)$. The set $\mathbb{H}(\pi)$ represents the set of concrete execution histories of π. The execution on each object should comply with the semantics of its type. The sub-history for each basic and linearizable object should be a member of the history semantics of that object type. (Eq. 15 and 16). The semantics is parametric in terms of the semantics of objects and they can be modularly defined.

Let $\mathcal{X} = (X, \sigma, \mathcal{L})$ and $X' = \sigma(X)$.
$\alpha_{\mathcal{X}} = \{\ obj \mapsto obj_{X'}, name \mapsto name_{X'}, thread \mapsto thread_{X'},$
$arg1 \mapsto arg1_{X'}, arg2 \mapsto arg2_{X'}, retv \mapsto retv_{X'},$
$initOf \mapsto initOf, commitOf \mapsto commitOf,$
$= \mapsto =,\ < \mapsto <,$
$exec \mapsto \lambda x.\ x \in X',\ \prec \mapsto \prec_{X'},\ \sim \mapsto \sim_{X'},\ \prec_o \mapsto \prec_{\mathcal{L}(\sigma(o))},$
$\ll \mapsto \ll_{X'},$
$o \mapsto \sigma(o),\ n \mapsto n,\ u \mapsto \sigma(u),\ \tau \mapsto \sigma(\tau),\ c \mapsto c,\ l \mapsto l\}$

The model relation $\models$:
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 $(\mathcal{X} \models \mathcal{A}_1) \wedge (\mathcal{X} \models \mathcal{A}_2)$,
$\mathcal{X} \models (\forall \ell : \mathcal{A})$ iff $\bigwedge_{l \in Labels(X')} (\mathcal{X} \models \mathcal{A}[\ell := l])$,
$\mathcal{X} \models (\forall t : \mathcal{A})$ iff $\bigwedge_{T \in Threads(X')} (\mathcal{X} \models \mathcal{A}[t := T])$.

Figure 8. Assertion Semantics

Assertion Semantics. For an execution $\mathcal{X} = (X, \sigma, \mathcal{L})$ and a logic assertion $\mathcal{A}$, we define the models relation $\mathcal{X} \models \mathcal{A}$ that states that the execution $\mathcal{X}$ models the assertion $\mathcal{A}$.

We define the domain as the set of constant labels and values. Given $\mathcal{X}$, we define the interpretation function $\alpha_{\mathcal{X}}$ that maps the variables, functions and predicates used in the assertion language to concrete values, functions and predicates on $\mathcal{X}$. The interpretation function $\alpha_{\mathcal{X}}$ and the models relation $\models$ are defined in Figure 8. Let $\mathcal{X} = (X, \sigma, \mathcal{L})$ and $X' = \sigma(X)$. The interpretation $\alpha_{\mathcal{X}}$ maps every variable x to $\sigma(x)$. It maps the predicate *exec* to $\lambda x.x \in X'$, the predicate $\prec$ to $\prec_{X'}$, the predicate $\prec_o$ to $\prec_{\mathcal{L}(\sigma(o))}$, and the function *obj* to $obj_{X'}$. The function $\alpha_{\mathcal{X}}$ is lifted to closed atomic assertions $\mathcal{R}$ by applying $\alpha_{\mathcal{X}}$ inductively to the structure of $\mathcal{R}$. An execution $\mathcal{X}$ models a closed atomic assertions $\mathcal{R}$ if $\alpha_{\mathcal{X}}(\mathcal{R})$ holds. The models relation for non-atomic assertions is inductively defined.

We lift the models relation to a description π. A description π models $\mathcal{A}$ (or $\mathcal{A}$ is valid for π), written as $\pi \models \mathcal{A}$, iff every execution of π models $\mathcal{A}$ i.e. $\forall \mathcal{X} \in [\![\pi]\!] : \mathcal{X} \models \mathcal{A}$. Similarly,

$\mathcal{T}_{TL2}$:

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

$\mathcal{D}_{TL2}$:

def $init_t()$

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

def $read_t(i)$

$R01 \triangleright$ $pv = wset[t].get(i)$,
if $(pv \neq \bot)$
$R02 \triangleright$ **return** pv,
$R03 \triangleright$ $t_1 = ver[i].read()$,
$R04 \triangleright$ $v = reg[i].read()$,
$R05 \triangleright$ $l = lock[i].read()$,
$R06 \triangleright$ $t_2 = ver[i].read()$,
$R07 \triangleright$ $s = rver[t].read()$,
if $(\neg(\neg l \ \wedge \ t_1 = t_2 \ \wedge \ t_2 \leq s))$
$R08 \triangleright$ **return** $\mathbb{A}$,
$R09 \triangleright$ $rset[t].add(i)$,
$R10 \triangleright$ **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** $(i \in wset[t])$
$C02_i \triangleright$ $l' = lock[i].trylock()$,
if $(\neg l')$
$C03_i \triangleright$ $lset[t].add(i)$
else
$C04_j \triangleright$ **foreach** $(j \in lset[t])$
$C05_{i,j} \triangleright$ $lock[j].unlock()$,
$C06_i \triangleright$ **return** $\mathbb{A}$,
$C07 \triangleright$ $wver = clock.iaf()$,
$C08 \triangleright$ $s = rver[t].read()$,
if $(wver \neq s + 1)$
$C09 \triangleright$ **foreach** $(i \in rset[t])$
$C10_i \triangleright$ $l = lock[i].read()$,
$C11_i \triangleright$ $cver = ver[i].read()$,
if $(\neg(\neg l \ \wedge \ cver \leq s))$
$C12_i \triangleright$ **foreach** $(j \in lset[t])$
$C13_{i,j} \triangleright$ $lock[j].unlock()$,
$C14_i \triangleright$ **return** $\mathbb{A}$,
$C15 \triangleright$ **foreach** $((i, v) \in wset[t])$
$C16_i \triangleright$ $reg[i].write(v)$,
$C17_i \triangleright$ $ver[i].write(wver)$,
$C18_i \triangleright$ $lock[i].unlock()$,
$C19 \triangleright$ **return** $\mathbb{C}$,
$\{C01 \rightarrow C07, \ C10 \rightarrow C11, \ C09 \rightarrow C15, C16 \rightarrow C17, \ C17 \rightarrow C18\}$,

$\mathcal{P}$: $tran_0, (tran_1 \parallel tran_2 \parallel ... \parallel tran_n)$

$\pi_{TL2} = (\mathcal{T}_{TL2}, \mathcal{D}_{TL2}), \mathcal{P})$

Figure 10. TL2 Algorithm Description

The marking $\sqsubseteq$ is the reflexive closure of $\sqsubset$.
The relation $\sqsubset$ is defined as follows:
$\forall t, t' : \ t \sqsubset t' \Leftrightarrow Eff(t) \prec_{clock} Eff(t')$
$\forall \ell_R, t : isTRead(\ell_R) \ \wedge \ isTWriter_i(t) \Rightarrow$
$\quad Let \ i = arg1(\ell_R)$:
$\quad t \sqsubset \ell_R \Leftrightarrow writeAcc_i(t) \precsim readAcc(\ell_R)$
$\quad \ell_R \sqsubset t \Leftrightarrow readAcc(\ell_R) \prec writeAcc_i(t)$
where

$$Eff(\tau) = \begin{cases} initOf(\tau)\text{'}I01 & \text{if } isAborted(\tau) \\ commitOf(\tau)\text{'}C07 & \text{if } isCommitted(\tau) \end{cases}$$

$readAcc(\ell_R) = \ell_R\text{'}R04$
$writeAcc_i(\tau) = commitOf(\tau)\text{'}C16_i$

Figure 11. TL2 Marking Relation $\sqsubseteq$

we define that a description π models Γ iff π models every assertion in Γ.

Soundness. The following soundness theorem states that if LSL deduces an assertion $\mathcal{A}$ for a description π using valid assumptions for π, then the deduced assertion $\mathcal{A}$ is valid for π as well. The proof is available in the appendix [28] § 15.2.

Theorem 4.1 (Soundness)**.**
$\forall \pi, \mathcal{A} : \ ((\pi, \Gamma \vdash \mathcal{A}) \ \wedge \ (\pi \models \Gamma)) \Rightarrow (\pi \models \mathcal{A})$.

5 TM Verification

We now state the correctness of transactional memory (TM) algorithms as an LSL assertion and apply LSL to prove the correctness of the TL2 [10] algorithm. The challenge is to verify that any concurrent execution of any set of well-formed transactions on TL2 is opaque. Markability factors out a large part of the proof, allows specification of the critical points of the algorithm and reduces verification to separate proof obligations about the order of these points. LSL inference rules can be easily used to prove the obligations based on the validation checks in the algorithm.

Transactional Memory. A TM object encapsulates a set of locations and provides four methods $init_t()$, $read_t(i)$, $write_t(i, v)$, and $commit_t()$. A well-formed transaction first calls $init_t()$ and then calls a sequence of $read_t(i)$ and $write_t(i, v)$ methods, and finally calls $commit_t()$. The method $commit_t()$ tries to commit transaction t and returns $\mathbb{C}$ (if it is successful). A TM object should detect if an inconsistency is about to happen between two concurrent transactions and should at least abort one of them. All methods may return abort $\mathbb{A}$ and terminate the transaction.

Correctness Assertion. We presented a decomposition called *markability* [31] of the correctness condition *opacity* [17]. Markability restates opacity in terms of three intuitive invariants. In Figure 9, we *state Markability as an assertion in LSL*. The markability assertion $isMarking(\sqsubseteq)$ is parametric with the marking relation $\sqsubseteq$. Figure 9.(a) represents preliminary definitions and Figure 9.(b) represents the marking assertion. We briefly explain markability. A TM algorithm is markable iff there exists a *marking* relation for it that is *write-observant*, *read-preserving*, and *real-time-preserving*.

A marking is a relation on the union of the transactions and the read method calls. We can think of the marking relation as the union of a collection of orders: (1) The *effect order*: The effect order is a total order of the transactions. The effect order represents the order in which the transactions appear to take effect, that is, the order that justifies the correctness of the execution. (2) The *access orders*: Let writers of location i be the committed transactions that have write method call(s) to i. Consider a read method call l_R that reads from a location i and doesn't abort. For each such l_R,

$$
\begin{aligned}
isInit(\ell) &\Leftrightarrow exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = init\\
isRead(\ell) &\Leftrightarrow exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = read\\
isWrite(\ell) &\Leftrightarrow exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = write\\
isCommit(\ell) &\Leftrightarrow exec(\ell) \wedge obj(\ell) = this \wedge name(\ell) = commit\\
isCommitted(\tau) &\Leftrightarrow \exists \ell : isCommit(\ell) \wedge thread(\ell) = \tau \wedge retv(\ell) = \mathbb{C}\\
isAborted(\tau) &\Leftrightarrow \exists \ell : exec(\ell) \wedge obj(\ell) = this \wedge thread(\ell) = \tau \wedge retv(\ell) = \mathbb{A}\\
isTRead(\ell_R) &\Leftrightarrow isRead(\ell_R) \wedge retv(\ell_R) \neq \mathbb{A}\\
isLocalTRead(\ell_R) &\Leftrightarrow isTRead(\ell_R) \wedge \exists \ell_W : isTWrite(\ell_W) \wedge arg1(\ell_R) = arg1(\ell_W) \wedge thread(\ell_R) = thread(\ell_W) \wedge \ell_W \prec \ell_R\\
isGlobalTRead(\ell_R) &\Leftrightarrow isTRead(\ell_R) \wedge \neg isLocalTRead(\ell_R)\\
isTWrite(\ell_W) &\Leftrightarrow isWrite(\ell_W) \wedge retv(\ell_W) \neq \mathbb{A}\\
isLocalTWrite(\ell_W) &\Leftrightarrow isTWrite(\ell_W) \wedge \exists \ell'_W : isTWrite(\ell'_W) \wedge arg1(\ell_W) = arg1(\ell'_W) \wedge thread(\ell_W) = thread(\ell'_W) \wedge \ell_W \prec \ell'_W\\
isGlobalTWrite(\ell_W) &\Leftrightarrow isTWrite(\ell_W) \wedge \neg isLocalTWrite(\ell_W)\\
isTWriter_i(\tau) &\Leftrightarrow \exists \ell_W : isTWrite(\ell_W) \wedge arg1(\ell_W) = i \wedge thread(\ell_W) = \tau \wedge isCommitted(\tau)
\end{aligned}
$$

(a) Preliminary Definitions

$$
\begin{aligned}
&isMarkingRel(\sqsubseteq) \Leftrightarrow\\
&\quad \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\\
&\qquad\qquad (t_1 \sqsubseteq t_2 \wedge t_2 \sqsubseteq t_3) \Rightarrow (t_1 \sqsubseteq t_3) \wedge\\
&\quad \forall \ell_R, t : Let\ i = arg1(\ell_R) : (isGlobalTRead(\ell_R) \wedge isTWriters_i(t)) \Rightarrow\\
&\qquad (\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}(\ell, \ell') \Leftrightarrow\\
&\quad \forall \ell'' : (isTWrite(\ell'') \wedge thread(\ell'') = t \wedge arg1(\ell) = i) \Rightarrow (\ell'' \preceq \ell \vee \ell' \preceq \ell'')\\
&NoWriterBet_i(q_1, \sqsubseteq, q_2) \Leftrightarrow \forall t : isTWriter_i(t) \Rightarrow t \sqsubseteq q_1 \vee q_2 \sqsubseteq t\\
&isLocalWriteObs \Leftrightarrow\\
&\quad \forall \ell_R : isLocalTRead(\ell_R) \Rightarrow Let\ t = thread(\ell_R), i = arg1(\ell_R) :\\
&\quad \exists \ell_W : isTWrite(\ell_W) \wedge thread(\ell_W) = t \wedge arg1(\ell_W) = i \wedge\\
&\quad \ell_W \prec \ell_R \wedge NoWriteBetween_{t,i}(\ell_W, \ell_R) \wedge retv(\ell_R) = arg2(\ell_W)\\
&isLastPreAccessor_{\sqsubseteq}(t', \ell_R) \Leftrightarrow\\
&\quad Let\ i = arg1(\ell_R), t = thread(\ell_R) : isTWriter_i(t') \wedge t' \sqsubseteq \ell_R \wedge t' \neq t \wedge NoWriterBet_i(t', \ell_R)\\
&isGlobalWriteObs(\sqsubseteq) \Leftrightarrow\\
&\quad \forall \ell_R : isGlobalTRead(\ell_R) \Rightarrow \exists \ell_W : isGlobalTWrite(\ell_W) \wedge Let\ t' = thread(\ell_W) :\\
&\qquad isLastPreAccessor_{\sqsubseteq}(t', \ell_R) \wedge arg1(\ell_R) = arg1(\ell_W) \wedge retv(\ell_R) = arg2(\ell_W)\\
&isWriteObs(\sqsubseteq) \Leftrightarrow isLocalWriteObs \wedge isGlobalWriteObs(\sqsubseteq)\\
&isReadPres(\sqsubseteq) \Leftrightarrow \forall \ell_R : isGlobalTRead(\ell_R) \Rightarrow Let\ i = arg1(\ell_R), t = thread(\ell_R) :\\
&\qquad\qquad NoWriterBet_i(\ell_R, \sqsubseteq, t) \wedge NoWriterBet_i(t, \sqsubseteq, \ell_R)\\
&isRealTimePres(\sqsubseteq) \Leftrightarrow \forall t, t' : t \not\preceq t' \Rightarrow t \sqsubseteq t'\\
&isMarking(\sqsubseteq) \Leftrightarrow isMarkingRel(\sqsubseteq) \wedge isWriteObs(\sqsubseteq) \wedge isReadPres(\sqsubseteq) \wedge isRealTimePres(\sqsubseteq)
\end{aligned}
$$

(b) Markability Assertion Parametric with $\sqsubseteq$

Figure 9. Markability

the access order is an antisymmetric relation that orders l_R and every writer of i. The access order represents where l_R's access to location i has happened between the accesses by the writers of i.

Write-observation requires that each read method call should read the most current value. Read-preservation requires that the location read by a read method call is not overwritten between the read accesses the location and the transaction takes effect. The real-time-preservation condition requires that the marking relation preserves the real-time order of transactions.

Algorithm Description. We have represented the TL2 algorithm [10] in our description language in Figure 10. The description π_{TL2} provides implementations of the four TM methods $init_t()$, $read_t(i)$, $write_t(i, v)$, and $commit_t()$ in the definitions section $\mathcal{D}$. The program in section $\mathcal{P}$ represents well-formed general client transactions. A client program

first runs $tran_0$ to initialize the shared variables and then concurrently runs n well-formed transactions $tran_1, .., tran_n$. A well-formed transaction t executes $init_t()$, then a sequence of $read_t(i)$ and $write_t(i, v)$ calls and finally a $commit_t()$; it finishes if it receives $\mathcal{A}$ from any call.

TL2 is a subtle algorithm. We briefly review how it works. TL2 uses the basic register $reg[i]$ to store the value of a location i. The algorithm reads $reg[i]$ at $R04$ and writes to $reg[i]$ at $C16$. Additionally, TL2 uses synchronization objects to help abort executions that would violate consistency. The idea is to give the value written in $reg[i]$ a *version number* that is stored in the $ver[i]$ register. TL2 uses a strong counter *clock*, whose value increases monotonically, to create such version numbers. Specifically, TL2 takes snapshots of *clock* both at $I01$ when a transaction starts and at $C07$ (with an increment-and-fetch operation, abbreviated *iaf*) during commit. TL2 validates the versions of read values before completing both the read and commit methods.

Verification. In Figure 11, we state the marking relation for the TL2 algorithm as an assertion in LSL. Intuitively, the effect order of transactions is the linearization order of their calls to *clock* at $I01$ and $C07$. The access order of read operations and writer transactions to location i is the execution order of their access to the $reg[i]$ register at $R04$ and $C16$. The following theorem states that the relation $\sqsubseteq$ defined above is a marking relation for TL2. The assertions Γ_0 are the properties of well-formed client transactions.

Theorem 5.1 (TL2 Correctness). $\pi_{TL2}, \Gamma_0 \vdash isMarking(\sqsubseteq)$.

We have mechanically checked the proof in PVS. The PVS theories for TL2 and Dekker are available [27].

6 Related Works

Manovit et al. [36] applied random testing to the TCC TM system. Lourenco et al. [33] reported several bugs during the porting of the TL2 algorithm. Given a TM algorithm and a bug pattern, our previous work [30] constructs a bug trace if the algorithm is prone to the bug pattern. It showed the incorrectness of algorithms that were deemed verified.

Although testing can find bugs, it does not prove their absence. To verify the correctness of TM algorithms, researchers have employed model checking and theorem proving. Model checkers from Cohen et al. [6, 7], and Guerraoui et al. [14–16] are the pioneering approach to verification of TM. Subsequently, the same approach was taken by O'Leary et al. [40] and Baek et al. [3]. Model checking can automate the verification process but it has been dependent on assuming properties about the TM algorithm and only scalable to a finite number of threads and locations or simplified algorithms. Later, Emmi et al. [12] tried to automatically infer algorithm invariants from small number of threads and memory locations. However, it worked on simplified algorithms due to scalability issues.

Attiya et al. [2] proved that opacity is sufficient for observational refinement of high-level atomic block semantics. Our previous work [31] showed the equivalence of opacity and markability and an informal proof of correctness for TL2. Koskinen and Parkinson presented a semantic model of serializability based on pulls from and pushes to an abstract shared log. Khyzha et al. [25] extended opacity to account for non-transactional accesses. In contrast to the current work, these works consider the correctness criteria, include only informal or non-mechanized proofs and do not include a logic and its soundness.

Singh [41] developed a runtime verification tool for TM algorithms. Although the tool is optimized with sound approximation techniques, the runtime overhead is still not negligible. Our previous work [29] presented a machine-checked theorem proving framework based on simulation between specifications and implementations [18, 34] represented as IOA [35] and verified the NORec algorithm [8]. Doherty et al. [11] adopted the same approach and proved the correctness of a pessimistic TM algorithm [37]. In follow-up works, Derrick et. al and Armstrong et al. [1, 9] simplified their simulation proofs by first model checking or proving the linearizability of the TM algorithm. In contrast to LSL that can reason about the algorithm description, these works require the algorithm to be translated to a transition system. In addition, they do not feature a logic.

To the best of our knowledge, LSL is the first logic that is applied to verification of transaction algorithms. In particular, it provides assertions for inter-thread execution and linearization orders that can directly capture the marking relation and the markability condition. Leveraging the proof of sufficiency of markability for opacity, verification of opacity is reduced to separate markability conditions that can be proved by the logic based on the validation checks in the algorithm. Logics based on concurrent separation logic [20, 39] and rely-guarantee reasoning [22] such as RGSep [43], LRG [13, 32], FCSL [38], GPS [42] and Iris [23, 24] require the specification of inter-thread relations as complicated global rely and guarantee conditions. Further, they need auxiliary variables even for the simple Dekker algorithm which may obscure the underlying design intuitions of the algorithms.

7 Conclusion

We presented a logic that supports syntactic reasoning about synchronization algorithm descriptions and features novel assertions and inference rules for execution and linearization orders. These assertions enable capturing critical orders between concurrent operations and in particular markability orders between transactions. We proved the soundness of the logic for a novel denotational semantics. We used the logic to machine-check a challenging proof of TL2.

References

[1] Alasdair Armstrong, Brijesh Dongol, and Simon Doherty. 2017. Proving opacity via linearizability: a sound and complete method. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 50–66.

[2] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. 2013. A Programming Language Perspective on Transactional Memory Consistency. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 309–318. https://doi.org/10.1145/2484239.2484267

[3] Woongki Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. 2010. Implementing and Evaluating a Model Checker for Transactional Memory Systems. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*. 117–126. https://doi.org/10.1109/ICECCS.2010.30

[4] M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. 2015. The problem of programming language concurrency semantics. In *ESOP*.

[5] John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative fence insertion. In *OOPSLA'15*. 367–385. http://doi.acm.org/10.1145/2814270.2814318

[6] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. 2007. Verifying Correctness of Transactional Memories. In *FMCAD*.

[7] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. 2008. Mechanical Verification of Transactional Memories with Non-transactional Memory Accesses. In *CAV*.

[8] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: streamlining STM by abolishing ownership records. In *PPoPP*.

[9] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. 2015. Verifying Opacity of a Transactional Mutex Lock. In *FM 2015: Formal Methods*, Nikolaj Bjørner and Frank de Boer (Eds.). Springer International Publishing, Cham, 161–177.

[10] D. Dice, O. Shalev, and N. Shavit. 2006. Transactional Locking II. In *DISC, (LNCS 4167)*.

[11] Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, and Heike Wehrheim. 2017. Proving opacity of a pessimistic STM. In *Leibniz International Proceedings in Informatics*, Vol. 70. Dagstuhl Publishing, 35–1.

[12] Michael Emmi, Rupak Majumdar, and Roman Manevich. 2010. Parameterized Verification of Transactional Memories. In *PLDI*.

[13] Xinyu Feng. 2009. Local rely-guarantee reasoning. In *POPL '09*.

[14] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. 2008. Model checking transactional memories. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 372–382.

[15] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2009. Software Transactional Memory on Relaxed Memory Models. In *CAV*.

[16] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2010. Model Checking Transactional Memories. *Distributed Computing* (2010).

[17] R. Guerraoui and M. Kapalka. 2008. On the Correctness of Transactional Memory. In *PPOPP*.

[18] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 449–465.

[19] M. P. Herlihy and J. M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *TOPLAS* 12, 3 (July 1990), 463–492.

[20] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle semantics for concurrent separation logic. In *ESOP*.

[21] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory *(LICS)*.

[22] Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83*, Vol. 9. 321–332.

[23] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, New York, NY, USA, 256–269. https://doi.org/10.1145/2951913.2951943

[24] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants As an Orthogonal Basis for Concurrent Reasoning. In *POPL '15*.

[25] Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. 2018. Safe privatization in transactional memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 233–245.

[26] Leslie Lamport. 1986. On Interprocess Communication. Part I: Basic Formalism. *Distributed Computing* 1, 2 (1986), 77–85.

[27] Mohsen Lesani. 2018. PVS Proof Theories. http://www.cs.ucr.edu/%7Elesani/companion/nfm19/PVSTheories.tar.gz. (2018).

[28] Mohsen Lesani. 2018. Submission Appendix. http://www.cs.ucr.edu/%7Elesani/companion/nfm19/Appendix.pdf. (2018).

[29] Mohsen Lesani, Victor Luchangco, and Mark Moir. 2012. A Framework for Formally Verifying Software Transactional Memory Algorithms. In *CONCUR*.

[30] Mohsen Lesani and Jens Palsberg. 2013. Proving Non-opacity. In *DISC, (LNCS 8205)*.

[31] Mohsen Lesani and Jens Palsberg. 2014. Decomposing Opacity. In *Distributed Computing*. Lecture Notes in Computer Science, Vol. 8784. 391–405.

[32] Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-fixed Linearization Points. In *PLDI '13*. 459–470.

[33] João Lourenc$_c$o and Gonc$_c$alo Cunha. 2007. Testing Patterns for Software Transactional Memory Engines. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD '07)*. ACM, New York, NY, USA, 36–42. https://doi.org/10.1145/1273647.1273655

[34] Nancy Lynch and Frits Vaandrager. 1992. Forward and Backward Simulations for Timing-based Systems. In *Proceedings of Real-Time: Theory in Practice (REX Workshop, Mook, The Netherlands, June 1991)*, J. W. de Bakker, W. P. de Roever, C. Huizing, and G. Rozenberg (Eds.). Springer-Verlag (*LNCS* 600), 397–446.

[35] Nancy A. Lynch and Mark R. Tuttle. 1989. An introduction to input/output automata. *CWI Quarterly* 2 (1989).

[36] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. 2006. Testing Implementations of Transactional Memory. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1152154.1152177

[37] Alexander Matveev and Nir Shavit. 2012. Towards a fully pessimistic stm model. (2012).

[38] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 290–310. https://doi.org/10.1007/978-3-642-54833-8_16

[39] Peter W. OHearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

[40] J. O'Leary, B. Saha, and Mark R. Tuttle. 2009. Model Checking Transactional Memory with Spin. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*. 335–342. https://doi.org/10.1109/ICDCS.2009.72

[41] Vasu Singh. 2010. Runtime Verification for Software Transactional Memories. In *Proceedings of the First International Conference on Runtime Verification (RV'10)*. Springer-Verlag, Berlin, Heidelberg, 421–435. http://dl.acm.org/citation.cfm?id=1939399.1939434

[42] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 691–707. https://doi.org/10.1145/2660193.2660243

[43] Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR*.