Sequential Verification of Serializability

H. Attiya  
Technion  
hagit@cs.technion.ac.il

G. Ramalingam  
Microsoft Research India  
grama@microsoft.com

N. Rinetzky  
Queen Mary University of London  
maon@dcs.qmul.ac.uk

Abstract
Serializability is a commonly used correctness condition in concurrent programming. When a concurrent module is serializable, certain other properties of the module can be verified by considering only its sequential executions. In many cases, concurrent modules guarantee serializability by using standard locking protocols, such as tree locking or two-phase locking. Unfortunately, according to the existing literature, verifying that a concurrent module adheres to these protocols requires considering concurrent interleavings.

In this paper, we show that adherence to a large class of locking protocols (including tree locking and two-phase locking) can be verified by considering only sequential executions. The main consequence of our results is that in many cases, the (manual or automatic) verification of serializability can itself be done using sequential reasoning.

Categories and Subject Descriptors  D.2.4 [Software Engineering]: Software/Program Verification; D.1.3 [Programming Techniques]: Concurrent Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms  Theory, Verification

Keywords  Concurrency, Verification, Serializability, Reduction, Modular Analysis

1. Introduction
In this paper, we study the dual problems of using sequential reasoning to verify that a concurrent module is serializable and the use of serializability to enable sequential reasoning of a concurrent module. A concurrent module encapsulates shared data with a set of procedures, which may be invoked by concurrently executing clients (threads). A key challenge in the design and analysis of concurrent modules is dealing with all possible interleavings of concurrently executing threads (i.e., procedure invocations whose executions overlap).

Serializability [5, 28] is a commonly desired and important criterion for concurrent modules. Informally, a concurrent module is said to be serializable if any (potentially interleaved) execution of a number of procedure invocations is equivalent to some sequential execution of those procedure invocations (one after another). One of the attractions of the serializability property is that it enables sequential reasoning: when a module is serializable certain properties can be verified by considering only non-interleaved executions of the module. This is equivalent to reasoning about the module assuming that it is used by a single-threaded (sequential) client.

One well-known way of ensuring serializability is to use locking protocols such as tree locking (TL) [15], two-phase locking (2PL) [23], or hand-over-hand locking, which is an instance of dynamic tree locking (DTL). If a concurrent module adheres to one of these protocols, then it is serializable, implying that a sequential reduction can be applied to the verification of other properties.

Sequential Reductions for Serializability. To the best of our knowledge, no prior work has addressed the question of whether sequential reductions apply to the problem of verifying that a module follows a locking protocol such as TL/2PL/DTL. Our results give a positive answer to this question.

Roughly stated, we establish that if the execution of every procedure satisfies the TL/2PL/DTL protocol in the absence of any interleaving of procedure executions then any interleaved execution of the procedures also satisfies the TL/2PL/DTL protocol. Informally, “sequential TL/2PL/DTL” implies “concurrent TL/2PL/DTL.” In fact, we generalize this result for a class of local locking protocols LP that ensure conflict-serializability [5]; this class includes TL and 2PL.

Our investigation of this question goes through the consideration of two types of sequential executions: A non-interleaved execution is an execution in which every procedure invocation is contiguous: i.e., there is no interleaving of instructions corresponding to different procedure invocations. An almost-complete non-interleaved execution is a non-interleaved execution in which all procedure invocations (except perhaps the last one) complete.

Our first reduction is to non-interleaved executions: we show that if the set of all non-interleaved executions of a module satisfy LP then the set of all executions of the module satisfy LP. Our second, and more complicated reduction, is to almost-complete non-interleaved executions: we prove that if the set of all almost-complete non-interleaved executions of a module satisfy LP then the set of all executions of the module satisfy LP, under an assumption that (roughly) requires each procedure invocation to complete when it executes solo.

Sequential Analysis of Serializable Modules. We next study the implication of our reductions for sequential analysis of a serializable module, e.g., to verify properties such as memory safety. While the idea that serializability (or atomicity) simplifies analysis is not new (e.g., see [13]), we establish several new results in this regard. We describe a class of properties, called transaction-local, whose verification can exploit a sequential reduction. We describe different types of sequential analyses, enabled by the different reduction theorems we establish, with differing preconditions. We also report on a preliminary evaluation of the effectiveness of the reduction techniques in verifying a couple of examples involving concurrent lists and trees.
Contributions. Our results enable modular and sequential reasoning, whether manual or automated, about concurrent modules both in verifying that a module adheres to a locking protocol and in development of algorithms for such modules. In the verification context, the reduction enables simpler and more efficient verification algorithms: e.g., it justifies the use of sequential Hoare Logic or sequential type systems or sequential abstract interpretation to verify that the module adheres to a locking protocol. Similarly, in the development context, a developer wishing to add, e.g., a new procedure to swap two adjacent elements in a list that uses hand-over-hand locking, does not have to worry about concurrent interleaving with other methods. The developer can rely on module invariants and think sequentially in designing the algorithm (as happens when using coarse-grained locking).

One of the implications of our result is that an automated tool [8] for synthesizing locking code to ensure the atomicity of the methods of a concurrent module can also use sequential reasoning if it utilizes locking protocols such as TL/2PL/DTL.

The result we present is a fundamental one about commonly used locking protocols. Readers might wonder if it follows directly from classical serializability theory, e.g., [5, 28]. However, the classical result is about a single execution and states that if, in an interleaved execution of transactions (procedure invocations, in our terminology), every transaction follows, e.g., TL, then this particular execution is serializable. We are, however, interested in checking whether a given module follows TL:i.e., we wish to check the antecedent of the above result for the set of all interleaved executions that can be produced by the module. The classical results do not let us simplify this question. (This was not a concern in databases, where transactions are governed by a centralized concurrency control monitor, which coordinates and controls their actions, at runtime, making sure they follow the locking protocol.)

Note. Due to space limitations, several proofs have been omitted from the paper and appear in [2].

2. The Challenge

We start with three illustrative examples shown in Fig. 1.

Example 1. Module T, shown in Fig. 1(a), follows the TL protocol. Informally, the TL protocol requires that accesses to every shared variable be protected by a lock, and that the locks be ordered in a forest. Locks should be acquired and released in a "hand-over-hand" order; a child lock is acquired before its parent is released.

(F or a more precise definition, see Sec. 4.2.)

Module T has a single procedure (τ) and three shared variables, Q, W, and E. Each variable also functions as its own lock. The tree-ordering on the shared variables is as follows: Q is the parent of W, which is the parent of E.

Let us first analyze sequential executions of this module: i.e., we consider executions where procedure τ may be invoked multiple times, but a new invocation of τ starts only after the previous invocation of τ terminates. We can verify that such executions satisfy the TL protocol. This verification, however, depends on certain invariants that hold in such sequential executions. E.g., verifying that the access to E happens only when the lock on E is held requires establishing that the second assertion (Q == W) holds. The latter depends on the fact that Q and W have the same value when read by procedure τ.

Concurrency, however, complicates the picture significantly. Invariants of sequential executions need not be invariants of concurrent executions. What do we need to do to verify that this module follows TL even in concurrent executions?

Our results show how we can use the sequential reasoning described above to verify that module T follows TL even in concurrent executions, inferring along the way that, from the viewpoint of every thread, Q == W always holds whenever a thread holds the locks of both variables, and that the other assertions embedded in the module are also correct in concurrent executions.

Example 2. Module DT implements a concurrent binary search tree following the hand-over-hand (lock-coupling) locking protocol. Fig. 1(b) shows only the delete procedure of the module.

Informally, the hand-over-hand protocol requires that every access to a field of a dynamically allocated object be protected by acquiring the lock of the object and that the heap be shaped like a tree/forest. (However, the forest shape requirement is allowed to be temporarily violated in the parts of the heap locked by a thread, which typically happens during heap updates.) Locks should be acquired and released in a "hand-over-hand" order. (For a more precise definition, see Sec. 6.1.)

Any sequential execution of module DT correctly follows the locking protocol. Concurrency, again, complicates the picture. The tree ordering of the locks is determined by the shape of the heap. Thus verifying that the module follows the locking protocol is inherently linked with a shape analysis of the heap. Note, however, that in concurrent executions there may be an unbounded number of threads simultaneously operating on different parts of the data structure. As a result, a concurrent shape analysis is typically more expensive and less precise than a sequential shape analysis.

We show that sequential reasoning can be used to verify that module DT follows DTL even in concurrent executions. In particular, this property can be verified by a sequential shape analysis.

Example 3. We now present an example where the use of sequential reasoning leads to results not valid for concurrent executions.

Fig. 1(c) shows module M which has two shared variables X and Y, and two procedures m and f. Let us check whether this module follows the 2PL protocol and also check whether the assertion "Y==0" in procedure f is true. Informally, the 2PL protocol requires that accesses to every shared variable be protected by a lock, and that after a lock is released, no lock is acquired (see Sec. 4.2).

Let us first consider only sequential executions. The assertion "Y==0" is true in the initial state of the module, and any complete execution of procedure m or procedure f preserves this invariant. As a result, the module satisfies 2PL.

However, this assertion does not hold in concurrent executions of M. Procedure m can, in fact, break this conjectured invariant, setting Y to be 1. When m does this, it also ends up in a non-terminating loop. As a result, the broken invariant is never exposed to a call of f in the sequential setting. In a concurrent setting, however, an invocation of f by another thread can observe this broken invariant. As a result, this invariant fails to hold for concurrent executions. When this invariant breaks, procedure f will no longer satisfy 2PL. Specifically, it ends up acquiring a lock after releasing a lock.

We show that if all procedures of the module are guaranteed to terminate then the sequential reduction described earlier is valid for verifying adherence to locking protocols such as 2PL.

Rather than checking for termination in all concurrent executions, we show that it suffices to check for termination only in sequential executions. This is weaker than solo-termination [12].

3. Preliminaries

In Sec. 3.1, we outline our programming model. In Sec. 3.2, we formalize the notions of executions and schedules. In Sec. 3.3, we review the notion of conflict-serializability and state some of its known properties.

3.1 Programming Model

We assume a programming language which allows for writing concurrent modules. For brevity, some of the standard technical details are omitted and can be found in [2].
Concurrent modules. A concurrent module encapsulates data that is private to the module and defines a set of procedures that may be invoked by clients of the module, potentially concurrently. The module variables are global to the module, i.e., shared by all procedure invocations. Procedures may also have local variables, which are private to the invocation of the procedures. Without loss of generality, we treat each invocation of a procedure as a thread.

Synchronization. Threads communicate using a shared memory comprised of global variables and an (unbounded) heap. Since procedure invocations may execute concurrently and access shared data, they utilize locks for concurrency control. A lock can be a global variable or a heap allocated object. Locks are exclusive of pro-

3.1.1 Syntax

Variables can be either of type boolean, integer, or a pointer to a user-defined type. We assume the syntactic domains $x \in V = V_\ell \uplus V_\gamma$ of variable identifiers, local variable identifiers $x, y \in V_\ell$, and global variable identifiers $X, Y \in V_\gamma$, $f \in F$ of field identifiers, $p \in P$ of procedure identifiers, and $\kappa \in \mathcal{K}$ of program points. We assume that variables, procedures, and program points have unique identifiers in every program.

Instead of committing ourselves to a particular syntax, we assume that body of a procedure is represented in a standard way using a control-flow graph. We refer to the vertices of a control-flow graph as program points $\kappa \in \mathcal{K}$. The edges of a control-flow graph are annotated with primitive instructions. Besides the following assumptions, the exact set of instructions is of no particular importance. We use assume statements to encode conditionals in the usual way, i.e., a control-flow edge is annotated with the statement “assume cond” to indicate that the branch is taken only if cond is true. Without loss of generality, we allow references to global variables and heap objects only in load/store instructions that copy values to and from local variables and acquire/release instructions. In particular, the condition of an assume statement cannot refer to a global variable.

3.1.2 Semantics

Memory states. Fig. 2 defines the semantic domains of memory states of module $m$ and the meta-variables ranging over them. We assume the semantic domain $t \in T$ of thread identifiers.

A memory state $\sigma = (g, h, q, \rho, \kappa, \ell, s, \sigma_\tau) \in \Sigma$ of a concurrent module $m$ is a triplet: $g$ is the global environment which records the values of the module-global variables, $h$ assigns values to fields of dynamically allocated objects. A value $v \in \mathcal{V}_\ell \mathcal{L}_\ell$ can be either a location, an integer, a boolean value, or null. $\rho$ associates a thread $t$ with its thread local state $g(t)$. A thread-local state $s = (\kappa, \rho, \ell) \in S$ is a triplet: $\kappa$ is the value of the thread’s program counter, $\rho$ records the values of its local variables, and $\ell$ is the thread’s lock set which records the locks that the thread holds. A lock can be either a global variable or a location.

Operational semantics. The behavior of a concurrent module can be described by a transition relation $\Sigma \times T \times (\mathcal{K} \times \mathcal{K}) \times \Sigma \uplus \{\bot\}$ that interlaces the execution of different threads. (\bot is a specially designated error state.) A transition $\Sigma \xrightarrow{\rho \ell \kappa} \Sigma'$ represents the fact that $\sigma$ can be transformed into $\sigma'$ via thread $t$ executing the instruction annotating control-flow edge $e$. Given a transition $\Sigma \xrightarrow{\rho \ell \kappa} \Sigma'$, we say that the transition is executed by $t$.

We present the (rather mundane) formal definition of transitions pertaining to a specific set of primitive instructions in [2].
able \(Y\). The statement “\(x.\text{acquire}()\)” attempts to acquire a lock on the heap object pointed to by local variable \(x\). The \(\text{assume}(b)\) instruction acts as \(\text{skipl}\) when executed on a state \(\sigma\) which satisfies \(b\). If \(b\) does not hold in \(\sigma\), the corresponding transition is not enabled. A memory fault (e.g., null-dereference) causes a transition to an error state.

### 3.2 Executions and Schedules

**Enabled execution step.** We refer to a pair \((t,e)\) consisting of a thread identifier and a control-flow edge as an execution step. We say that \((t,e)\) is enabled in state \(\sigma\) if there exists some transition \(\sigma \xrightarrow{e} \sigma'\) and that it is disabled otherwise. Assume that the program-counter of \(t\) is at the source edge \(e\) (in state \(\sigma\)). Then, \((t,e)\) is disabled in \(\sigma\) if \(e\) is labeled with either an assume statement whose condition evaluates to false or an acquire statement of a lock that is currently held by another thread.

**Executions.** An execution \(\pi\) is a sequence of transitions \(\sigma_0 \xrightarrow{\tau_1} \sigma_1 \ldots \sigma_k\) that starts at the initial state (i.e., \(\sigma_0 = \sigma_0\)) with the target of every transition being the same as the source of the next transition. (When discussing executions, we may omit the labels on the transitions if no confusion is likely.)

The sub-execution of thread \(t\) in an execution \(\pi\) is the subsequence of transitions executed by thread \(t\) in \(\pi\).

**Transaction.** A transaction \(T_i\) executed by a thread \(t_i\) in an execution \(\pi\) is the sequence of control-flow edges annotating the transitions executed by \(t_i\). Note that a transaction is a path in the control flow graph of the procedure \(p\) invoked by \(t_i\), starting at \(p\)'s entry node.

**NI-executions.** An execution is non-interleaved (abbreviated NI-execution) if instructions of different threads are not interleaved, i.e., for every pair of threads \(t_i \neq t_j\), either all the transitions executed by \(t_i\) come before any transition executed by \(t_j\), or vice versa.

**ACNI-executions.** A thread is said to have completed in an execution if its program counter in the final state is at the exit node of the corresponding procedure. An execution is complete if every thread in its initial state has completed. An execution is complete non-interleaved (abbreviated CNI-execution) if it is both complete and non-interleaved. An execution is almost-complete non-interleaved (abbreviated ACNI-execution) if it is non-interleaved, and all but the thread executing last has completed.

**Schedules.** A schedule \(\Phi = \{t_1,e_1\} \ldots \{t_k,e_k\}\) is a sequence of execution steps. \(\Phi\) is valid if for every thread identifier \(t\), the subsequence of edges paired with \(t\) comprise a path in a control flow graph of a procedure \(p\) starting at \(p\)'s entry node. \(\Phi\) is feasible if it is induced by some execution \(\pi\), i.e., \(\pi = \sigma_0 \xrightarrow{\{t_1,e_{\pi \sigma_0}\}} \sigma_1 \ldots \{t_k,e_{\pi \sigma_k}\} \sigma_\pi\).

### 3.3 Conflict Serializability

**Well locked executions.** An execution is well-locked if a thread never accesses a global variable or a field of a dynamically allocated object without holding its protecting lock. (In database terminology, well-lockedness is referred to as “obeying the access rules” [5].) In this paper, we assume that every global variable also acts as its own protecting lock and that every dynamically allocated object also acts as the protecting lock of its own fields.

**Conflict serializability.** Given an execution, we say that two transitions conflict if (i) they are executed by two different threads, (ii) they access some common global variable or a heap allocated object. The usual definition of conflict requires at least one of the conflicting instruction to be a write (to the global variable or object). However, this simpler definition suffices for us, since we use exclusive locks for reading as well. (Note that a lock acquire/release instruction is considered to be a write of the corresponding lock.)

Executions \(\pi_1\) and \(\pi_2\) are conflict-equivalent if they include the same transactions and they agree on the order between conflicting transitions. (I.e., the \(i\)th transition of a thread \(t\) precedes and conflicts with the \(j\)th transition of thread \(t'\) in \(\pi_1\) iff the former precedes and conflict with the latter in \(\pi_2\).) Conflict-equivalence ensures view equivalence [5], i.e., every transaction in conflict-equivalence executions reads and writes the same sequence of values to and from the global variables.

An execution is conflict-serializable if it is conflict-equivalent with a non-interleaved execution. A module is conflict-serializable if all of its executions are conflict-serializable.

Since conflict-equivalent executions produce the same state, we have the next lemma.

**Lemma 1.** Let \(\pi\) and \(\pi'\) be conflict-equivalent (well-locked) executions. (a) For any thread \(t\), the set of locks held by \(t\) after \(\pi\) is the same as the set of locks held by \(t\) after \(\pi'\). (b) The execution \((t,e)\) is enabled after \(\pi\) iff it is enabled after \(\pi'\).

The next lemma is handy for NI-reduction, as it allows to move a step “earlier” in the execution.

**Lemma 2.** Let \(\pi_w = \pi_1 \pi_2 \pi_3\) be a well-locked NI-execution, where \(\pi_1\) is the sub-execution by a thread \(t\). If the execution step \((t,e)\) is enabled after \(\pi_w\), then it is also enabled after \(\pi_1\).

### 4. NI-Reductions for Static Protocols

We wish to verify that a given module is conflict-serializable, by verifying that it adheres to a locking protocol, e.g., TL or 2Pl, that guarantees conflict-serializability. However, we would like to do this by considering only the module’s almost-complete non-interleaved executions rather than all the executions it can generate.

We take a two-step approach towards our goal. The first step is a reduction to non-interleaved executions (NI-reduction): we show for a class of locking protocols that if every NI-execution of the module satisfies the protocol then all executions of the module satisfies the protocol. The second step is a reduction to almost-complete non-interleaved executions (ACNI-reduction): we show for a class of locking protocols that if every ACNI-execution of the module satisfies the protocol then all executions of the module satisfies the protocol.

We first establish our reductions in a static setting where we assume that the program does not use the heap (dynamic memory allocation) or pointers. This restriction is just a convenience in the case of 2Pl: the reductions carry over even in the presence of dynamic memory allocation. However, extending the reductions to handle dynamic memory allocation is more challenging in the case of TL because this requires changes to the protocol itself. Section 6 presents our reductions for Dynamic Tree Locking (DTL).

#### 4.1 Transaction-Local Properties

Given a memory state \(\sigma = (\gamma, h, d)\), the thread-owned state of thread \(t\), defined by \(\text{owned}_t(\sigma) = (\gamma, t, h, d, (t))\) where \(d(t) = (k, p, L)\) is its own thread-local state in \(\sigma\) and the projection of the global state on the locks that \(t\) holds.

The thread-owned view of a thread \(t\) executing a transaction \(T\) in an execution \(\pi\) is obtained by replacing every state \(\sigma\) in the corresponding \(t\)'s sub-execution by \(\text{owned}_t(\sigma)\).

We now define a certain class of properties of executions. To avoid restricting ourself to any particular logic we simply identify a property \(\phi\) of executions with the set of executions that satisfy the property. Similarly, we identify a property \(\phi\) of thread-owned views with the set of thread-owned views that satisfy the property.
We say that a property $\phi$ of executions is transaction-local if (a) there is a property $\phi'$ of thread-owned views such that an execution $\pi$ is in $\phi$ iff all the thread-owned views of all the threads launched in $\pi$ are in $\phi'$, and (b) $\phi$ is prefix-closed: if $\pi$ is in $\phi$, then every prefix of $\pi$ is in $\phi$. For any transaction-local property $\phi$ of executions, we will use $\phi_T$ to denote the property $\phi'$ of thread-owned views guaranteed to exist by the above definition.

4.2 Local Conflict Serializable Locking Protocols

A local conflict-serializable locking protocol (abbreviated LCS-locking-protocol) is a transaction-local property that guarantees well-lockedness and conflict-serializability. In other words, a property $LP$ of executions is an LCS-locking-protocol iff (i) $LP$ is transaction-local and (ii) $\pi \in LP$ implies that $\pi$ is well-locked and conflict-serializable. A module follows a locking protocol $LP$ if every possible execution of the module satisfies $LP$.

There are two important instances of LCS-locking-protocols. Two-phase locking (2PL) requires each execution $\pi$ to be well-locked and to ensure that every thread executes in two phases: a growing phase, in which the thread may acquire locks but not release any locks, followed by a shrinking phase, in which the thread may release locks, but may not acquire locks. If an execution follows 2PL then it is conflict-serializable [23].

Tree locking (TL) assumes that locks are organized in the shape of a forest (collection of trees). An execution $\pi$ follows the TL policy if it is well locked and every thread (i) can acquire a lock only if it holds the lock on its parent in the tree, unless the lock is the first acquired by the transaction, and (ii) does not acquire a lock that it has previously released. If an execution follows TL then it is conflict-serializable [15].

4.3 NI-Reduction for LCS-Locking-Protocols

**LEMMMA 3.** Let $LP$ be an LCS-locking-protocol. Let $\pi_{ni} = \pi_{p} | \pi_{\overline{g}}$ be a non-interleaved execution, where $\pi_{p}$ is the sub-execution of a thread $t$. Let $\pi_{ni} \xrightarrow{t,e} \sigma$ be an execution that extends $\pi_{ni}$ by a single transition. If $\pi_{p} | \pi_{\overline{g}}$ satisfies $LP$ and $\pi_{p} \xrightarrow{t,e} \sigma'$ satisfies $LP$, then $\pi_{p} | \pi_{\overline{g}} \xrightarrow{t,e} \sigma'$ also satisfies $LP$.

**THEOREM 4.** If every NI-execution of a module satisfies an LCS-locking-protocol $LP$, then every execution of the module satisfies $LP$.

**Proof:** The proof is by induction on the length of $\pi$. The base case is when the trace is empty, which is immediate.

For the induction step assume that $\pi' = \pi \xrightarrow{t,e} \sigma'$, where $\sigma'$ is the state produced by a thread $t$ executing an instruction $st$ annotating $e$. By the induction hypothesis, execution $\pi$ follows the locking protocol. The locking protocol guarantees conflict-serializability. Thus, there exists a non-interleaved execution $\pi_{ni}$ which is conflict-equivalent to $\pi$.

By Lemma 1, the execution of $(t,e)$ is enabled after $\pi_{ni}$ as well. Let $\pi_{ni}$ be of the form $\pi_{p} | \pi_{\overline{g}} | \pi_{s}$ where $\pi_{s}$ is the sub-execution by thread $t$. By Lemma 2, $(t,e)$ is also enabled after $\pi_{p} | \pi_{s}$.

Since $\pi_{p} | \pi_{s} | (t,e)$ is an NI-execution, it satisfies $LP$ (by assumption). By Lemma 3, $\pi_{ni} | (t,e)$ satisfies $LP$ as well. Since $\pi$ satisfies $LP$, and $\pi$ and $\pi_{ni}$ are conflict-equivalent (and produce the same state), it follows that $\pi' = \pi \xrightarrow{t,e} \sigma'$ satisfies $LP$ as well.

Since TL and 2PL are LCS-locking-protocols, this theorem shows that a module can be verified to follow TL or 2PL by considering only its NI-executions.

5. ACNI-Reductions for Static Protocols

In this section, we show how to improve upon NI-reduction. We would like to show that all executions of a concurrent module satisfy an LCS-locking-protocol $LP$ if and only if all ACNI-executions of the module satisfy $LP$.

Unfortunately, this goal is too ambitious, because of the possibility of procedure invocations that do not terminate. As the example in Fig. 1(c) shows, it is possible to have a procedure invocation $T$ whose changes to shared data are visible to other procedure invocations even though $T$ will never terminate. This can lead to execution behaviors that can never be observed in an almost-complete non-interleaved execution.

Therefore, we will settle for a slightly weaker result. We say that an ACNI-execution of a module is complete if it is the prefix of a complete non-interleaved execution (CNI-execution) of the module. We will show that ACNI-reduction holds for a module as long as every ACNI-execution of the module is complete. (One way to show that every ACNI-execution is complete is to use a sequential termination analysis to show that the last procedure invocation of an ACNI-execution is guaranteed to terminate in the absence of any other concurrent procedure invocation. Our ACNI-reduction theorem can also be used directly, without a termination analysis, for any concurrent module where every loop exit edge is labelled with an “assume true” statement, as discussed in Section 7.)

There is a second hurdle we need to consider, namely that of deadlock. Even if all transactions are guaranteed to terminate in isolation, they may end up in a deadlock in an interleaved execution. We directly show that this is not a concern for 2PL and TL and discuss the general case in Sec. 5.4.

5.1 Proof Strategy

We take a two step approach. First, we establish that when all ACNI-executions satisfy $LP$ and are complete then all NI-executions satisfy $LP$. Then, our NI-reduction theorem (The. 4) let us conclude that all executions satisfy $LP$.

**DEFINITION 1.** Let $\pi_{ni}$ be an NI-execution with a schedule $\alpha_1 \cdots \alpha_h$, each $\alpha_i$ representing the sub-schedule of a different thread $t_i$. Let $\pi_{ni}$ be a CNI-execution with a schedule $\alpha_1 | \beta_1 | \cdots | \alpha_j | \beta_j$, each $\alpha_i, \beta_i$ representing the sub-schedule of a different thread $t_i$. We say that $\pi_{ni}$ is an equivalent completion of $\pi_{ni}$ if $\pi_{ni}$ satisfies the following condition: for all $i < j$, the set of variables accessed by the execution of $\beta_i$ and $\alpha_j$ are disjoint.

The next lemma is a simple consequence of the definitions.

**LEMMMA 5.** Let $\pi_{ni}$ be an equivalent completion of $\pi_{ni}$. For any thread $t$, the thread-owned view of $\pi_{ni}$ is a prefix of the thread-owned view of $\pi_{ni}$.}

Our overall proof strategy is to take an NI-execution $\pi_{ni}$ and construct an equivalent completion $\pi_{ni}$. By Lemma 5, if $\pi_{ni}$ satisfies $LP$, then $\pi_{ni}$ satisfies $LP$ as well. We now present an iterative scheme that will successfully construct an equivalent completion for some, but not necessarily every, NI-execution $\pi_{ni}$.

Let the schedule of $\pi_{ni}$ be $\alpha_1 \cdots \alpha_k$, each $\alpha_i$ representing the sub-schedule of a different thread $t_i$. We construct a sequence of schedules $\gamma_{ih}$ where $1 \leq h \leq k$, where each $\gamma_i$ is of the form $\alpha_1 | \beta_1 | \cdots | \alpha_i | \beta_i$ and is the schedule of an CNI-execution. Furthermore, the execution of $\beta_i$ is guaranteed to not access any variable accessed by any $\alpha_j$ for $j > i$.

We let $\gamma_0$ be the empty schedule. We construct the schedule $\gamma_{i+1}$ from $\gamma_i$ as follows. Note that $\gamma_i | \alpha_{i+1}$ is the schedule of an ACNI-execution. Hence, by assumption, this is completeable. That is, there exists an CNI-execution $\pi_{i+1}$ with a schedule $\gamma_i | \alpha_{i+1} | \beta_{i+1}$.

If the execution of $\beta_{i+1}$ in $\pi_{i+1}$ does not access any variable accessed by $\alpha_j$ in $\pi_{i}$ for any $j > i + 1$, then we define $\gamma_{i+1}$ to be $\gamma_i | \alpha_{i+1} | \beta_{i+1}$. Otherwise, our algorithm fails.
If the algorithm succeeds at every step \( i \), then we define the execution \( \pi_k \) produced finally to be the iterative completion of \( \pi_n \). Otherwise, we say that the iterative completion of \( \pi_n \) is not defined.

**Lemma 6.** If the iterative completion of an NI-execution \( \pi_n \) is defined, then it is an equivalent completion of \( \pi_n \).

It follows from the above discussion that the key open issue in proving ACNI-reduction is handling the case where \( \beta_{i+1} \) accesses some variable accessed by some \( \alpha_j \) for \( j > i + 1 \) in the above algorithm for constructing the iterative completion. There are two reasons why this “failure” case may arise, one illustrated by TL and one illustrated by 2PL. We show how to work around them and prove ACNI-reduction for both these protocols. We then combine these ideas to establish ACNI-reduction for any LCS-locking-protocol satisfying a property we call progressive (see Sec. 5.4).

The following lemma shows that extending certain executions by a single transition does not violate an LCS-locking-protocol L.P.

**Lemma 7.** Assume that every ACNI-execution of a module satisfies an LCS-locking-protocol L.P. Let \( \pi \) be an NI-execution that has an equivalent completion. Then, any NI-execution \( \pi' = \pi \tau \) consisting of \( \pi \) following by a single transition \( \tau \) satisfies L.P.

**Proof:** Let transition \( \tau \) be generated by thread \( t \) executing instruction \( e \). Since \( \pi \) satisfies L.P., the sub-execution of every thread in \( \pi \) satisfies L.P. Thus, to prove that \( \pi' \) satisfies L.P. we just need to verify that the sub-execution of \( t \) in \( \pi' \) satisfies L.P.

Let \( \alpha \) denote the sub-execution of \( t \) in \( \pi' \). Thus, the sub-execution of \( t \) in \( \pi' \) is \( \alpha t \). Let \( \pi_{\alpha t} \) be an equivalent completion of \( \pi \) which exists by assumption. Consider the ACNI-execution \( \pi' \) which is the prefix of \( \pi_{\alpha t} \) that ends with \( \alpha \). (We let \( \pi' = \pi_{\alpha t} \) if \( \alpha \) is empty.) We can show that the thread-owned state of \( t \) at the end of this ACNI-execution is the same as that at the end of \( \pi \). Since \( t \) can execute instruction \( e \) after \( \pi \), it follows that it can execute the same instruction after \( \pi' \) as well. (See Lemma 1.) The sub-execution of \( t \) in \( \pi' \) extended by \( e \) must satisfy L.P. (since this extension is an ACNI-execution). Hence, \( \alpha \tau \) must also satisfy L.P.

It follows that \( \pi' \) satisfies L.P.

**5.2 ACNI-reduction for TL**

We first establish that ACNI-reduction is valid for TL. The inductive proof carries a stronger property, to help in the induction step.

**Theorem 8.** If every ACNI-execution of a module satisfies TL and is complete, then every NI-execution \( \pi \) of the module (a) satisfies TL and (b) has an equivalent completion \( \pi_{\text{em}} \).

**Proof:** Assume that every ACNI-execution of the module satisfies TL and is complete. We prove (a) and (b) by induction on the length of the execution. The base case (an empty execution) is trivial. Assume as the inductive hypothesis that \( \pi \) satisfies (a) and (b). Consider an NI-execution \( \pi' = \pi \tau \sigma' \), where \( \sigma' \) is the state produced by the execution of an instruction \( e \) by a thread \( t \) after \( \pi \). (a) follows immediately from Lemma 7 and the inductive hypothesis.

We now prove (b). Our strategy is to use the algorithm for constructing the iterative completion of an NI-execution described earlier. However, the iterative completion of \( \pi' \) may not be defined, as illustrated by the following simple example. Assume that \( \pi' \) consists of a thread \( t_1 \) acquiring a lock on \( u \), followed by a thread \( t_2 \) acquiring a lock on \( v \), a child of \( u \) (in the lock-ordering forest). Suppose the next step of \( t_1 \) is to acquire a lock on \( v \). If so, the iterative completion of \( \pi' \) is not defined. The trick in this case is to reorder the execution of \( t_1 \) and \( t_2 \) in \( \pi' \) to obtain a conflict-equivalent execution \( \pi'' \) and construct the iterative completion of \( \pi'' \). We now show that this can always be done.

Let the schedule of \( \pi' \) be \( \alpha_1 \cdots \alpha_n \), each \( \alpha_i \) representing the sub-schedule of a different thread \( t_i \). We assume, without loss of generality, that every thread \( t_i \) acquires at least one lock in \( \pi' \). (A thread \( t_i \) that does not acquire any locks can be easily handled.) Consider the state \( \sigma_i \), produced by the execution \( \pi_i \), of \( \alpha_1 \cdots \alpha_i \). Let \( \text{locked}_i \) denote the set of variables locked by \( t_i \) in \( \pi_i \). Let \( \text{lockable}_i \) denote the set of children of variables in \( \text{locked}_i \). If not already been locked by \( t_i \) in the execution \( \pi_i \). These are the variables that \( t_i \) can immediately lock, if available, without violating TL. Let \( \text{lockable}_i \) denote the set of descendants of variables in \( \text{lockable}_i \) (where a vertex is considered to be its own descendant). These are the variables that \( t_i \) can eventually lock without violating TL.

Given \( i < j \), we say that a thread \( t_j \) may-depend-upon \( t_j \) if the first variable locked by \( t_j \) (in \( \pi' \)) is contained in \( \text{lockable}_i \). These are the dependences that can create us problem while constructing an iterative completion.

We identify a permutation of the set of threads \( t_k \) to \( t_k \) in which \( t_j \) occurs after \( t_j \) if \( t_k \) may-depend-upon \( t_j \). We do this iteratively as follows. Initially, we start with an empty sequence \( S_0 \). Given \( S_j \), a permutation of \( t_j \) through \( t_j \), we identify \( S_{j+1} \) as follows. If none of the \( t_i \) in \( S_j \) may-depend-upon \( t_j \), we append \( t_j \) to \( S_j \) to obtain \( S_{j+1} \). Otherwise, we find the first occurrence of a \( t_i \) in \( S_j \) such that \( t_i \) may-depend-upon \( t_j \) and insert \( t_j \) just before \( t_i \) to obtain \( S_{j+1} \). Let the final sequence \( S_k \) be \( t_1, \ldots, t_k \).

Consider the schedule \( \phi_{t_1} \cdots \phi_{t_k} \). We claim that this produces an NI-execution \( \pi'' \) that is conflict-equivalent to \( \pi' \). Furthermore, we claim that the iterative completion of \( \pi'' \) is defined and that this yields an equivalent completion for \( \pi' \) as well. Proofs omitted due to lack of space.

**5.3 ACNI-Reduction for 2PL**

We will now present an analogue of the above theorem for 2PL. The challenge in constructing an iterative completion of an NI-execution that follows 2PL is the possibility of a deadlock. (TL, on the other hand, guarantees deadlock-freedom.)

Given two threads \( t_1 \) and \( t_2 \) in an execution \( \pi \), we say that \( t_2 \) depends on \( t_1 \) if \( t_1 \) acquires a global variable \( g \) during the execution that is accessed later by thread \( t_2 \). We define the schedule slice of \( \pi \) with respect to thread \( t_1 \), to be the subsequence of the schedule of \( \pi \) consisting of steps taken by \( t_2 \) or any other thread \( t' \) on which \( \pi \) depends (directly or transitively). It follows trivially that the schedule slice is a feasible schedule. We define the slice of \( \pi \) with respect to \( t_1 \), denoted \( \pi_{t_1} \), to be the execution induced by the schedule slice. For any thread \( t' \) that occurs in \( \pi_{t_1} \), it can be shown that the thread-owned view is the same in \( \pi \) and \( \pi_{t_1} \).

**Theorem 9.** If every ACNI-execution of a module satisfies 2PL and is complete, then (a) every NI-execution \( \pi \) of the module satisfies 2PL, and (b) for every NI-execution \( \pi \) of the module and every thread \( t, \pi_{t_1} \) has an equivalent completion \( \pi_{t_1} \).

**Proof:** The proof is by induction on the length of the execution. Consider an NI-execution \( \pi' = \pi \tau \sigma' \), where we assume that \( \pi \) satisfies (a) and (b). We show that \( \pi' \) also satisfies (a) and (b).

(a) Since \( \pi \) satisfies 2PL, we just need to show that \( \pi \) satisfies 2PL. Consider \( \pi_{t_1} \), which must be of the form \( \pi_{t_2} \tau \sigma' \). It follows from Lemma 7 and the inductive hypothesis that \( \pi_{t_1} \) follows 2PL. Therefore, \( \pi' \) itself satisfies (a) (i.e. follows 2PL).

(b) We now argue that the iterative completion of \( \pi_{t_1} \) is defined. Assume that we take a schedule \( \alpha_1 \cdots \alpha_n \) and construct the schedule \( \alpha_1 \beta_1 \cdots \alpha_n \beta_n \) using our iterative completion algorithm. We just need to show that the set of variables accessed by \( \beta_j \) is disjoint from the set of variables accessed by \( \alpha_j \) for \( j > i \). Note that the execution of the thread \( t_i \) (\( i < k \)) is included in \( \pi_{t_1} \) only if some other thread depends on \( t_i \) in \( \pi' \). Such a dependence is
In this section, we present the Dynamic Tree Locking (DTL) protocol. The graph contains the edge representing the shared program state using a graph. Each global variable and each heap-allocated object is represented by a vertex in the graph. The graph contains the edge $\pi \rightarrow v$ if the variable/object represented by $v$ points to the object represented by $v$. If $u$ represents a structured variable/object whose $f$ field points to $v$, then this is represented by a labelled edge $u \xrightarrow{f} v$. In the sequel, we will omit the edge labels when no confusion is likely. We note that the target of an edge must always represent a heap object. In the sequel, we will often abuse notation and not distinguish between a vertex and the variable/object it represents.

We define the set of vertices in the scope of a thread $t$ at any point during an execution as follows: Initially, the scope of any thread is empty. Whenever a thread $t$ acquires a lock on $u$, all successors of $u$ (in the graph representation) become part of $t$'s scope. Whenever a thread $t$ allocates a new object $u$, $u$ becomes part of $t$'s scope. Whenever a thread $t$ releases a lock on $u$, all successors of $u$ are removed from $t$'s scope.

Note that in comparing states produced by different executions, the address of a heap allocated object is irrelevant. Instead, equivalence of states is established with respect to a bijective mapping between the heap allocated objects of the two states, in the obvious way. More generally, given two executions $\pi_1$ and $\pi_2$, we define the correspondence relation between heap objects of the two executions as follows: the object allocated by the $i$-th execution-step of thread $t$ in $\pi_1$ corresponds to the object allocated by the $i$-th execution-step of thread $t$ in $\pi_2$ (when both exist). We use this correspondence to compare states in the two executions.

**Definition 2.** We say that an execution satisfies the Dynamic Tree Locking Protocol (DTL) if it satisfies the following conditions:

1. The execution is well-locked.
2. A thread never acquires a lock on an object $u$ after it has released a lock on that object $u$.
3. The first lock a thread acquires is on a global variable.
4. Subsequent to acquiring its first lock, a thread can acquire a lock on a vertex only if it is in the thread's scope.
5. A thread can insert an edge $u \xrightarrow{v}$ only if $v$ is in the thread's scope. (More precisely, the thread can assign the address of an object represented by vertex $v$ to a global variable or an object's field only if $v$ is in the thread's scope.)
6. Whenever a thread releases a lock on $u$, for every successor $v$ of $u$, $v$ must be $v$'s only predecessor. I.e., $u \rightarrow v$ and $w \rightarrow v$ implies $u = w$. (In other words, whenever a thread releases $v$ from its scope, $v$ must have a unique predecessor.)

Note that a thread is allowed to create a new resource at any point.

**6.2 Properties of DTL**

**Lemma 11.** The following properties hold true at any point during and just after any execution satisfying DTL:

1. A vertex not in the scope of any thread has at most one predecessor.
2. All predecessors of a vertex in the scope of thread $t$ must currently be locked by $t$.
3. A vertex can be in the scope of at most one thread.

**Proof:** By induction. Note that when a new vertex is created it belongs to its creator's scope. When a thread releases $u$ from its scope, it ensures that $u$ has at most one predecessor. A vertex not in the scope of any thread enters the scope of a thread $t$ when $t$ acquires a lock on $u$'s unique predecessor. Once $t$ enters $t$'s scope, no other thread $t'$ can insert any edge $x \rightarrow u$ or acquire $u$ for its scope: for $t'$ to insert an edge $x \rightarrow u$, $u$ must first be in $t'$'s scope, and for $u$ to enter $t'$'s scope $u$ must first have an unlocked predecessor that $t'$ can lock. The result follows.

Since DTL conditions 1 to 5 are transaction-local, we have the next lemma:

**Lemma 12.** Let the thread-owned views of thread $t$ in two executions $\pi$ and $\pi'$ be the same. Let the execution of step $(t, e)$ be enabled after both $\pi$ and $\pi'$. Then, the execution of step $(t, e)$ after $\pi$ satisfies DTL conditions 1 to 5 if the execution of step $(t, e)$ after $\pi$ satisfies DTL conditions 1 to 5.
However, condition 6 of the DTL protocol is not a transaction-local property. This is why our earlier reductions do not hold for DTL and we must separately establish the following results.

**Lemma 13.** Let $\pi = \pi_1\pi_2\ldots\pi_s$ be a non-interleaved execution, where $\pi_i$ is the sub-execution of a thread $t$. Let $\pi_i \xrightarrow{}^\delta \pi$ be an execution that extends $\pi_i$ by a single transition. If $\pi_1\pi_2\ldots\pi_s$ satisfies DTL and $\pi_1 \xrightarrow{}^\delta \pi'$ satisfies DTL, then $\pi_1\pi_2\ldots\pi_s \xrightarrow{}^\delta \pi$ also satisfies DTL.

**Proof:** Note that the thread-owned view of $t$ is the same in $\pi_1\pi_2\ldots\pi_s$ and $\pi_1\pi_2\ldots\pi_s$. Since the step $(t, e)$ satisfies DTL after $\pi_1\pi_2\ldots\pi_s$, it follows from Lemma 12 we just need to check for condition 6.

Assume that step $(t, e)$ releases a lock on $u$, and we have an edge $u \rightarrow v$ in the state after $\pi_1\pi_2\ldots\pi_s$. Thus, $v$ must be in the scope of $t$ throughout the execution of $\pi_s$. Consequently, $v$ must have the same set of predecessors after $\pi_1\pi_2\ldots\pi_s$ as after $\pi_1\pi_2\ldots\pi_s$. Thus condition 6 is satisfied by $\pi_1\pi_2\ldots\pi_s \xrightarrow{}^\delta \pi'$ if it is satisfied by $\pi_1\pi_2\ldots\pi_s \xrightarrow{}^\delta \pi$.

**Lemma 14.** Let $\alpha_1\beta_1\gamma_1\delta_1$ denote the schedule of an NI-execution, where $\beta_1$ and $\delta_1$ denote the schedules executed by threads $u$ and $t$ respectively. Assume that no transitional conflict-dependence exists between $u$ and $t$ in this execution. If $\alpha_1\beta_1\gamma_1\delta_1$ both satisfy DTL, then so does $\alpha_1\beta_2\gamma_2\delta_2$. (In other words, $\beta_2$ does not affect the correctness of $\delta_2$.)

**Proof:** Let $\pi_1$ denote the execution of $\alpha_1\beta_1\gamma_1\delta_1$. We need to show that the execution of all instructions in $\delta_1$ in $\pi_1$ satisfies DTL. Let $\pi_2$ denote the execution of $\alpha_2\beta_2\gamma_2\delta_2$. The absence of any transitional conflict-dependence between $\beta_1$ and $\delta_1$ in $\pi_1$ means that Lemma 12 applies and that we just need to verify that the execution of $\delta_1$ in $\pi_1$ satisfies condition 6. This follows inductively. The key properties to note are: (a) The subgraph induced by the set of all vertices accessed by $\delta_1$ and their successors at any point during the execution of $\delta_1$ in $\pi_1$ is isomorphic to the corresponding subgraph induced in $\pi_2$. (b) Furthermore, none of the vertices in these subgraphs have any predecessor outside the subgraph. As a result a vertex going out of $t$'s scope at some point in $\pi_1$ can have more than one predecessor if the corresponding vertex going out of $t$'s scope at the corresponding point in $\pi_2$ has more than one predecessor. The result follows.

### 6.3 NI Reduction for DTL

**Theorem 15.** If every NI-execution of a module satisfies DTL, then every execution of the module satisfies DTL.

**Proof:** The proof is identical to that of Theorem 4, except for the use of Lemma 13 in place of Lemma 3.

### 6.4 ACNI Reduction for DTL

We first establish that the iterative completion algorithm described in Sec. 5.1 will succeed under a simple condition.

**Lemma 16.** Let $\alpha_1\alpha_2\ldots\alpha_k$ be the schedule of an ACNI-execution $\pi$, each $\alpha_i$ represents the schedule of a different thread $t_i$. Let $\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_k\beta_k$ be the schedule another ACNI-execution $\pi'$, where $\alpha_i\beta_i$ represents the schedule of thread $t_i$. Assume that $t_i$ acquires at least a single lock in execution $\pi$. For any $i < j$, the set of locations accessed (read, written, or locked) during the execution of $\beta_i$ in $\pi'$ is disjoint from the set of locations accessed during the execution of $\alpha_j$ in $\pi$. (Note that we compare heap locations in different executions modulo the correspondence relation described in Sec. 6.1.)

**Proof:** Let $\sigma_i$ be the state produced by the execution of $\alpha_1\ldots\alpha_i$. Let $V_i$ denote the set of vertices in $\sigma_i$. Let scope denote the set of vertices in the scope of $t_i$ in state $\sigma_i$. Let reachable, denote the set of vertices in $\sigma_i$ that are reachable via some path from some vertex in scope.

For $j > i$, the execution of $\alpha_j$ in $\pi$ cannot lock any vertex in reachable. We can establish this inductively. Thread $t_j$ cannot lock any vertex in the scope of any other thread $t_k$ in $\sigma_k$. Furthermore, any vertex $y$ not in the scope of any thread in $\sigma_i$ must have at most one predecessor $x$. If $t_j$ is unable to lock $x$, it will be unable to lock $y$ as well.

Now consider the execution of $\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_k\beta_k$. Let us identify any vertex allocated in a step $s$ in $\pi$ with the location allocated by the corresponding step $s'$ in $\pi'$. We claim that any vertex in $V_i$ that is locked by $t_i$ during the execution of $\beta_i$ must be in reachable, this follows inductively, since for $t_i$ to acquire a lock on any $y$ it must first hold a lock on some predecessor $x$ of $y$.

In particular, this implies that the set of vertices locked by $\beta_j$ and $\alpha_j$ must be disjoint (where $i < j$).

**Theorem 17.** If every ACNI-execution of a module satisfies DTL and is completeable, then (a) every NI-execution $\pi$ of the module satisfies DTL, and (b) every NI-execution $\pi$ of the module has an equivalent completion $\pi_{eq}$ (which is a CNI-execution, by definition).

**Proof:** Assume that every ACNI-execution of the module satisfies T1 and is completeable. We prove (a) and (b) by induction on the length of the execution. The base case (an empty execution) is trivial. Assume the inductive hypothesis that $\pi$ satisfies (a) and (b). Consider an NI-execution $\pi' = \pi \xrightarrow{}^2 \pi'$, where $\pi'$ is the state produced by the execution of an instruction $e$ by a thread $t$ after $\pi$. We assume that $\pi$ has a transition conflict-dependence on all other threads that execute in $\pi$. (Otherwise, we can omit the execution of any other thread $u$ that $t$ does not have a dependence on from $\pi'$ to get a shorter execution $\pi''$. From the inductive assumption, $\pi''$ must satisfy DTL.)

Let $\pi_{eq}$ be an equivalent completion of $\pi$. $\pi_{eq}$ must satisfy DTL by assumption, since it is an ACNI-execution. Let the schedule of $\pi$ be $\alpha_1\alpha_2\ldots\alpha_k$, and the schedule of $\pi_{eq}$ be $\alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_k\beta_k$, where each $\alpha_i$ (and $\beta_i$) represents execution by a thread $t_i$. Note that Lemma 16 applies to $\pi$ and $\pi_{eq}$.

Case 1: First, consider the case where $t \neq t_k$. Thus, $(t, e)$ is the first step performed by a new thread $t$. We first show that $(t, e)$ is enabled after $\pi_{eq}$ as well. Suppose that $(t, e)$ tries to acquire a lock on $u$. Since $(t, e)$ is enabled after $\pi$, $u$ must be unlocked after $\pi$. Furthermore, we assumed the existence of a conflict-dependence between $t_k$ and $t$. This conflict-dependence can exist only if $t_k$ acquired and then released a lock on $u$ (during $\alpha_k$ in $\pi$). The definition of an equivalent completion implies that $t_k$ acquires and releases a lock on $u$ during $\alpha_k$ in $\pi_{eq}$. Hence, $t_k$ cannot acquire a lock on $u$ again during $\beta_k$. Hence, $u$ must be unlocked after $\pi_{eq}$ as well. Thus, $(t, e)$ is enabled after $\pi_{eq}$.

Now $\pi_{eq}$, $t \xrightarrow{}^2 \pi'$ is an ACNI-execution, which satisfies DTL by assumption. So, the execution of $(t, e)$ following $\pi_{eq}$ satisfies DTL. As a result, the execution of $(t, e)$ after $\pi$ satisfies all local properties of DTL. The only non-local property we need to check is condition 6. But the first instruction of a thread $t$ cannot release a lock. (Otherwise, the ACNI-execution $\pi_{eq} \xrightarrow{}^2 \pi'$ would violate DTL.) Hence, (a) follows.

As for (b), the ACNI-execution $\pi_{eq}$, $t \xrightarrow{}^2 \pi'$ must have a completion by assumption. This gives us an equivalent completion for $\pi'$. (Lemma 16 above ensures that this is an equivalent completion.)

Case 2: Now, consider the case where $t = t_k$. Consider the schedule $\gamma = \alpha_1\beta_1\alpha_2\beta_2\ldots\alpha_k\beta_k$. (We don’t consider $\beta_k$ since $t_k$ continues executing.) We now show that $(t, e)$ is enabled after $\gamma$. Assume that $(t, e)$ acquires a lock on $u$ (following execution $\pi$). If...
u is the first lock acquired by \( t_1 \), then the reasoning is the same as in case 1. Otherwise, \( t_j \) must hold a lock on the unique predecessor \( x \) of \( u \) after \( \pi \). This implies that none of the \( \beta_j \), for \( j < k \), can acquire a lock on \( x \) in the execution \( \gamma \). This follows from Lemma 16 (or, equivalently, from the definition of an equivalent completion). As a result, none of the \( \beta \) can acquire a lock on \( u \) either. Hence, \( u \) remains unlocked at the end of \( \gamma \). Hence, \( (t, e) \) is enabled after \( \gamma \).

Now \( \gamma(t,e) \) is an ACNI-execution, which satisfies DTL by assumption. We need to now show that \( \pi(t,e) \) satisfies DTL. By Lemma 12, we just need to check for non-local property 6. This follows just as in the proof of Lemma 14. Hence, (a) follows.

As for (b), the ACNI-schedule \( \gamma(t,e) \) must have a completion by assumption. This gives us an equivalent completion for \( \pi' \). (By Lemma 16 this completion is equivalent.)

7. Sequential Analyses for Concurrent Modules

Our reduction theorems enable the use of sequential analyses to verify that a module adheres to a locking protocol such as, e.g., DTL and is conflict-serializable. For a conflict-serializable module, our results also enable the use of sequential analysis to infer or verify other properties of the module. In this section we make these claims more precise by describing different kinds of sequential analyses and the conditions under which they can be used and formalizing the types of properties that can be verified using these analyses.

7.1 Verifiable Properties

**Transaction-Local Properties.** We first establish that analogues of our earlier reduction theorems hold for verifying transaction-local properties. Since transaction-local properties observe only what happens within each thread, the next theorem is a straightforward consequence of known properties of conflict-serializability.

**Theorem 18.** Let \( M \) be a conflict-serializable module and let \( \phi \) be a transaction-local property. If every NI-execution of module \( M \) is well-locked and satisfies property \( \phi \), then every execution of the module satisfies \( \phi \).

The following theorem shows that ACNI-reductions hold for transaction-local properties.

**Theorem 19.** Let \( M \) be a conflict-serializable module and let \( \phi \) be a transaction-local property. If every ACNI-execution of module \( M \) is well-locked, completable and satisfies \( \phi \), then every execution of \( M \) satisfies \( \phi \).

The above theorems justify the use of sequential analyses (described later) to infer or verify transaction-local properties. As an example of properties that can be so verified, consider the module \( T \) shown in Fig. 1(a). The property \( Q \Rightarrow W + 1 \) in the first assert statement is transaction-local and can be verified by a sequential analysis. Similarly, a sequential analysis can also establish the second assert statement which, as it is concerned only with local variables, is also transaction-local. Note, however, that the property \( Q \Rightarrow W \) is not a transaction-local property at the program point of the second assertion, as \( Q \)'s lock has been released.

**Module Invariants.** A program state \( \sigma \) is quiescent if there exists a complete execution that produces \( \sigma \). A property \( \phi \) of program states is said to be a module invariant if all quiescent states satisfy \( \phi \). Our results also justify the use of sequential analyses to infer module invariants. For example, a sequential analysis of module \( T \), shown in Fig. 1(a) can determine that \( Q \Rightarrow W \) is a module invariant. Module invariants are of particular interest since one can reason about module procedures assuming that they are invoked in a state satisfying the module invariants (when the module is conflict-serializable and its ACNI-executions are completable). Note that the inferred module invariants do not have to be transaction-local, but can still be used in the reasoning as explained above.

7.2 Sequential Modular Analysis

We describe several sequential modular analyses and the conditions under which these analyses can be used. The term **modular analysis** refers to analysis of a given module, independent of any specific client of the module. The information computed by a modular analysis is valid for any possible client of the module.

**Optimistic Analysis.** An optimistic analysis exploits ACNI-reduction and can be used when the conditions for ACNI-reduction are met. Briefly, an optimistic analysis amounts to an analysis of the program obtained by combining the module with its most-general sequential client. The most-general sequential client of module DT is shown in Fig. 3(a). Thus, an optimistic analysis iteratively identifies the module invariants, and analyzes the invocation of every module procedure on any state satisfying the module invariant. We start with a tentative module invariant that characterizes the initial state of the module’s data (before any procedure is invoked), and iteratively analyze the module’s procedures when invoked in a state satisfying the tentative module invariant, and weaken the module invariant to account for the state that can be produced at the end of the module procedure.

**Completable Analysis.** The use of an optimistic analysis or ACNI-reduction requires verifying that every ACNI-execution is completable. This requirement can be proved with termination analysis (e.g. [6, 10, 16]). Note that the termination analysis needs to only establish that a procedure invocation terminates when invoked in a state satisfying the module invariant and executes in isolation. In other words, the termination analysis itself can be optimistic.

However, completability is a weaker requirement than termination, as illustrated by the following examples. Contrast the problematic code snippet from Fig. 1(c) acquire(Y); Y = 1; release(Y); while true;

with acquire(Y); Y = 1; release(Y); while(+);. Both snippets are potentially non-terminating. However, the second one is “completable” according to our definition. Thus, an optimistic analysis is valid in the second, although it is not valid in the first case.

The two requirements differ only for non-deterministic programs. This is quite significant, as many analyses typically do not interpret branching or looping conditions. Such analyses can effectively use an optimistic approach without requiring a termination analysis. For the reduction to hold, we still need to guarantee that every ACNI-execution can be completed. However, instead of verifying this, if we weaken conditions on branches (assume statements) to ensure that this holds trivially; we are fine. Stated differently, given a module \( M \), we weaken conditions in \( M \) to construct a module \( M_w \) that has more behaviors than \( M \) and one that trivially satisfies the “completable” condition. Our theorem justifies the use of optimistic analysis for \( M_w \). Because \( M_w \) has more behaviors than \( M \), a successful verification of \( M_w \) applies to \( M \) also.

**Pessimistic Analysis.** This approach analyzes each procedure of the module independently once. It assumes nothing about the initial state (of the module’s global variables) in which the procedure is invoked. However, the analysis will assume that the procedure is executing in isolation, without interference from any concurrent execution of other procedure invocations. This analysis relies only on NI-reduction. Hence, it does not require verifying that all ACNI-executions are completable.

A pessimistic analysis has the potential to be more efficient than an optimistic analysis but is less precise. (E.g., note that pessimistic analysis cannot verify either of the assertions of module \( T \).
We implemented an optimistic analysis of hand-over-hand (lock-coupling) algorithms for sorted lists and binary search trees. Our analysis was derived from the sequential analyses of [17] and [20] and implemented using TVLA [18]. Our analysis successfully verified that these algorithms do not dereference null-valued pointers, do not leak memory, and follow the DTLS locking policy.

The list analysis also verified that the list remains acyclic and sorted. The tree analysis also verified that procedures maintain a forest of trees which are heap-sorted. The analyzer also verifies that every procedure invocation in sequential execution terminates (as required by ACNI-reduction). The termination argument is based on the size-change argument [3, 16]: It shows that in every iteration of the set of reachable elements from the procedure variables is a strict subset of the reachable elements in the previous iteration. We are not aware of any other automatic verification tool that is capable of verifying partial and total correctness of the lock-coupling tree algorithm.

We measured the additional cost required by our extensions to the sequential analysis of the lock-coupling tree. (The analysis of the lock-coupling list is too short for a meaningful comparison.) Tab. 1(b) compares the cost of our analysis (⋆) with a vanilla version that verifies all of the above properties except for adherence to the locking protocol. The results show that the cost of the two analyses is in the same order of magnitude. The difference between the two is essentially the cost paid to validate that the sequential invariants are also concurrent invariants.

9. Discussion and Related Work

The idea of simplifying reasoning about concurrent programs using reduction, i.e., by treating code fragments that satisfy appropriate conditions atomically, is quite old. However, not many have considered the problem of whether the process of reduction itself can be done efficiently by considering only the reduced system. One exception is the work by Stoller and Cohen [25]. They consider the aforementioned issue and present sufficient conditions under which the reduction itself can benefit from reduction. Our results are a refined version of [4, 21], and are able to verify that the list coupling algorithm is linearizable for an unbounded number of threads. The analysis of [27] is a separation logic based analysis which uses rely guarantee reasoning. This analysis is the cheapest analysis in our benchmark. However, it is based on an abstraction that does not track sortedness. As a result, this analysis fails to verify linearizability in some cases.

Table 1. Experimental Results. Experiments performed on a machine with a 2.4 G Hz Intel Core 2 Duo processor and 2 Gb memory running version 10.5.7 of the Mac OS X operating system.
of (automatically) identifying if a code fragment satisfies these conditions.

Flanagan and Qadeer [13] present an algorithm for identifying procedures (and code fragments) that behave atomically, exploiting Lipton’s theory. The Flanagan-Qadeer approach decomposes the atomicity checking problem into two parts: (A) the algorithm they present identifies atomic regions of code under the assumption that the system does not have any dataraces and (B) they use existing algorithms to check if the system has any potential datarace.

Our results show that atomicity checking (via checking adherence to a locking protocol such as 2PL) can be simpler than the sub-problem of checking for dataraces in the following sense. Absence of dataraces and the stronger property of well-lockedness (which requires that every shared variable resp. field of a dynamically allocated object be accessed by a thread t only when t holds the variable’s resp. field’s protecting lock) do not enjoy sequential reduction. In other words, it is possible that all threads satisfy well-lockedness when there is no interleaving of thread executions, but an interleaved execution may not satisfy well-lockedness, as illustrated below.

Example 4. Consider module S shown in Fig. 3(b). Module S is well-locked in all NI-executions, but not in all interleaved executions. In any NI-execution, the conditional assignment of 2 to y in procedure p never executes because the condition b is always false at this point. However, in an interleaved execution, q might change the value of b to 1, which causes the conditional assignment of 2 to y to happen (when procedure p does not have a lock on y), violating the well-lockedness condition.

The analogue of our approach, in the Flanagan-Qadeer setting, would be to do part (B) assuming atomicity of code identified as atomic by part (A), which can make the second check more efficient and/or more precise. Of course, such a cyclic dependence (assume-guarantee reasoning) would have to be justified as being correct, which is one of the main contributions of our work.

Code following the TL or DTL protocol cannot be identified as being atomic by the Flanagan-Qadeer approach. Furthermore, the Flanagan-Qadeer approach does not apply when atomicity depends on other invariants maintained by the module which must be simultaneously established to verify atomicity (such as in cases involving list/tree manipulation), unlike our approach.

Partial order reduction (POR) techniques [9] combat the state-explosion problem by exploring only a representative subset of all program executions. In general, however, verifying that a subset of all executions is representative is hard and requires analysis that accounts for all interleaved executions.

Elmas et al. [11] present a proof technique for concurrent programs that combines reduction with abstraction, in an iterative fashion, to simplify verification. The core reduction is based on Lipton’s theory of movers.

10. Conclusions and Future Work

Verifying that a concurrent module is serializable is an important and challenging problem. In this paper, we exploit the fact that often concurrent modules guarantee serializability through a local locking protocol to establish a reduction of the aforementioned concurrent verification problem to a sequential verification problem.

Our results enable (automatic or manual) analysis and verification tools for concurrent modules to perform the analysis more efficiently by considering only sequential executions. Currently, we assume that all locks are exclusive. In the future, we plan to also consider non-exclusive (shared) read locks as well as other concurrency control mechanisms, like those based on timestamps [26].

Abstractly our work may be thought of as a “staged rely-guarantee reasoning”. We take a specification for the procedures (namely, the locking protocol) and we apply rely-guarantee reasoning to analyze the specifications themselves (rather than an implementation of the specification). The results we so establish guarantee that, as a second stage, one can verify that the procedures satisfy their specifications (the locking protocols) in a sequential setting and get, for free, the proof that the procedures satisfy their specifications in a concurrent setting. We believe that generalizing this approach (of staged rely-guarantee reasoning) to other problems is an interesting direction for future research.

Acknowledgements: We are grateful to Ahmed Bouajjani, Moody Sagiv, and Eran Yahav for collaborating with us in the early stages of this research, and for many interesting discussions. We thank Rachid Guerraoui, Vasu Singh, and the anonymous referees for their useful comments.

References

### A. ACNI-reduction for Progressive Locking Protocols

In this section, we show that ACNI-reduction is valid for any progressive LCS-locking-protocol.

**Lemma 20.** Let LP be any progressive LCS-locking protocol. Assume that every ACNI-execution of a module satisfies LP and is completeable. Let $\pi_n$ be any non-interleaved execution that satisfies LP. If all threads except the one executing last in $\pi_n$ are visible then $\pi_n$ has an equivalent completion.

**Proof:** Let the set of the given NI-execution $\pi_n$ be of the form $\alpha_1 \alpha_2 \ldots \alpha_k$, where each $\alpha_i$ denotes the sub-schedule of a thread $t_i$. We essentially wish to let every thread $t_i$ run to completion, producing sub-schedule $\gamma_i = \alpha_i \beta_i$. However, while doing so, we need to ensure that the new instructions $\beta_i$ do not modify (lock) any global variable that is read by $\alpha_j$ of any subsequently executing thread. In order to achieve this, we may need to re-order the sub-schedules of some threads, while preserving the dependencies between them. We now sketch how we can do this. Recall that threads $t_i$ and $t_j$ are said to conflict in $\pi_n$ if they access some common global variable. If $t_i$ and $t_j$ conflict, and $i < j$, we say that $t_j$ is dependent on $t_i$.

We now show how to complete the given NI-execution iteratively. We start with the empty schedule $\Phi_0$, and construct the schedule $\Phi_{i+1}$ from $\Phi_i$ by identifying the thread $t_{\alpha_i}$ to be scheduled next, constructing a thread-completion $\gamma_{\alpha_i} = \alpha_j \beta_j$, and defining $\Phi_{i+1} = \Phi \gamma_{\alpha_i}$. Let $S_i$ denote the set $\{s_1, \ldots, s_i\}$. We maintain the following invariant as we construct the schedule: $\Phi_i$ is a feasible schedule; $\Phi_i$ preserves dependencies among the scheduled threads (i.e., if we have thread $t_{\alpha_i}$ is dependent on thread $t_{\alpha_j}$ in $\pi_n$, then we ensure that $p < q$); further, in the execution of $\Phi_i$, the instructions executed by $\beta_j$ will not reference (or lock) any global variable referenced by $\alpha_j$. For any $j$ not in $S_i$, the any thread that has not been scheduled yet.

Given schedule $\Phi_i$, we say that a thread $t_j$ has been completed if $j \in S_i$. We say that a thread $\alpha_i$ is a candidate (for scheduling next) if all threads it depends on (in execution $\pi_n$) have completed. We note that at least one candidate must exist since the dependence edges between threads form an acyclic graph. The invariants maintained guarantee that for any candidate $t_j$, $\Phi_i \alpha_j$ is feasible and produces an ACNI-execution where the thread-owned view of $t_j$ is the same as in $\pi_n$. Furthermore, our assumption that all ACNI-executions can be completed imply that there exists a CNI-execution with a schedule $\Phi_i \alpha_j \beta_j$. Our assumptions also guarantee that this new schedule satisfies the locking protocol LP. If $\beta_j$ (in the above CNI-execution) does not reference (lock) any location referenced by any thread not in $S_i \cup \{t_j\}$, we say that $t_j$ is a valid candidate.

If a valid candidate $t_j$ exists, we define $s_i$ to be $j$ and $\Phi_{i+1}$ to be $\Phi_i \alpha_j \beta_j$.

All that remains is to show that at least one valid candidate must exist. We prove this by contradiction. Assume that no valid candidates exist. We will derive a contradiction as below. The informal idea is to show that if all candidates are "blocked" then we can reach a configuration that the locking protocol is not supposed to allow (either a deadlock among visible transactions or an execution that is not conflict-serializable).

As shown above, every candidate thread $t_j$ has a potential completion $\alpha_j \beta_j$. Since the candidate is not valid, $\beta_j$ must have a prefix of the form $\beta'_j \epsilon_j$, where $\beta'_j$ does not reference any "forbidden variable" (any variable referenced by a thread not in $S_i \cup \{t_j\}$, while $\epsilon_j$ references a forbidden variable. Assume that $\epsilon_j$ references a forbidden variable also referenced by $\alpha_j$, where $t_j \not\in S_i \cup \{t_j\}$. We say that $t_j$ is blocked by $t_j$, and say there is a blocking dependence from $t_j$ to $t_j$. The key to note is that there must exist a cycle involving the union of blocking dependence edges and normal dependence edges. (The cycle can be constructed since every candidate thread has at least one incoming blocking dependence edge, while every non-candidate (unscheduled) thread has at least one incoming dependence edge.)

Let $\delta_j$ denote $\alpha_j \beta'_j$. Since $\alpha_j \beta_j$ satisfies LP (as mentioned above), it must be well-locked and hence $\epsilon_j$ must be an acquire instruction. (Well-lockedness also implies that no other non-scheduled thread can access a variable for which $t_j$ holds the lock at the end of $\alpha_j$; thus, the forbidden variables cannot include any variable for which $t_j$ holds the lock at the end of $\alpha_j$.)

Let $t_{x_1}, \ldots, t_{x_n}$ be the set of candidate threads. Let $t_{x_1}, \ldots, t_{x_n}$ be the set of remaining unscheduled threads in an order satisfying the dependencies between them. Consider the following completion $\Phi' = \Phi \delta_{x_1} \ldots \delta_{x_n} \delta_{x_{n+1}} \ldots \delta_{x_k}$. This is a feasible schedule that satisfies LP. Let $\Phi'$ denote the abrupt completion of this schedule with respect to every $t_{x_j}$. This is also a schedule that follows the locking protocol LP. Note that this schedule may not be possible in our concurrent module, but we can still use it in our argument below.

From this point on, we iteratively choose a candidate thread $t_{x_j}$ that is trying acquire a (forbidden) lock that is not currently held by another candidate. We let $t_{x_j}$ acquire the forbidden lock, and then we force it to terminate abruptly (by releasing all its locks). Eventually, either all candidates would have acquired its forbidden lock or we must get stuck because of a deadlock involving the candidates. The latter is not possible according to definition of a progressive locking protocol. The former is also not possible because the above process converts every blocking dependence that existed originally into a real dependence. As a result, the final execution will have a cycle of dependencies among the transactions, while the execution is supposed to be conflict-serializable (since it follows the locking protocol). The result follows. ■

**Proof:** (Theorem 10) The proof is by induction on the length of $\pi$. The claim is trivial for the base case, when $\pi$ is empty. Consider the inductive step. Part (a) follows from our inductive hypothesis and Lemma 7. From part (a), it follows that $\pi$ and, hence, $\pi'$ are conflict-serializable. Let $\pi_n$ be the NI-execution obtained by conflict-serialization of $\pi'|$. Part (b) follows by applying Lemma 20. ■