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.

POPL'10, January 17–23, 2010, Madrid, Spain.

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 to a module 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. For brevity, several proofs and results have been omitted from the main body of the paper and appear in the appendices, including, in particular, reduction theorems for the *Dynamic DAG Locking* protocol (DDL).

2. The Challenge

We start with three illustrative examples shown in Figure 1.

EXAMPLE 1. Module `T`, shown in Figure 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. (For a more precise definition, see Section 4.2).

Module `T` has a single procedure (`t`) 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 `t` may be invoked multiple times, but a new invocation of `t` starts only after the previous invocation of `t` 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 `t`.

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. Figure 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 Section 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.

Figure 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 Section 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.

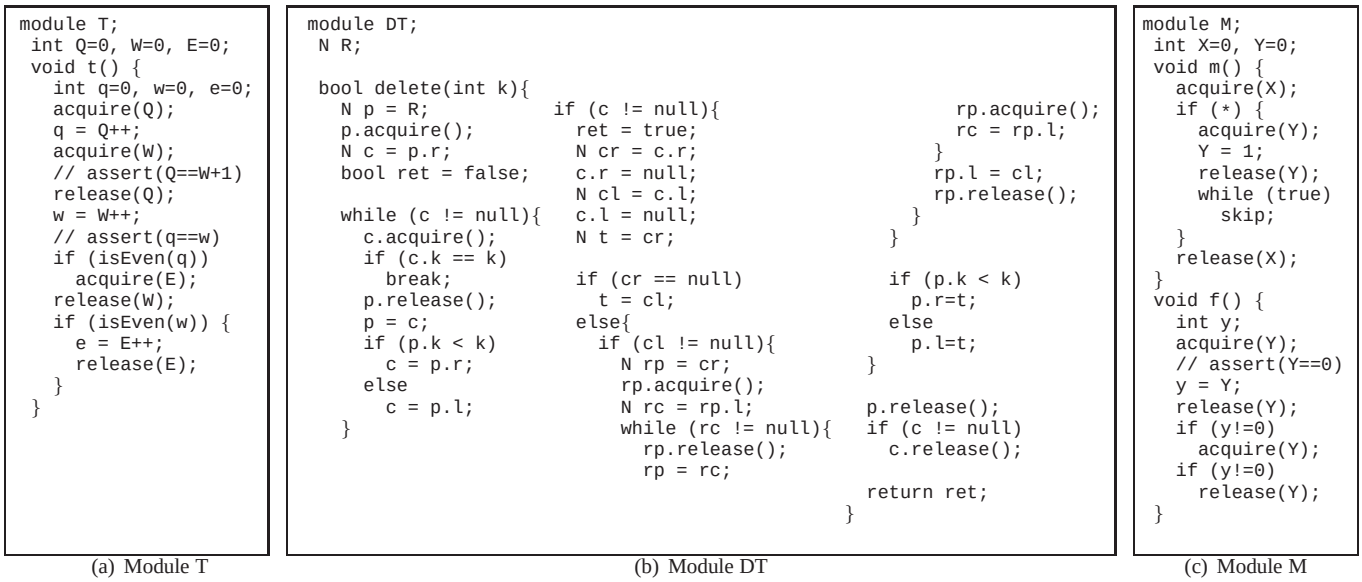
However, this assertion does not hold in concurrent executions of `M`. Procedure `m` can, in fact, break this conjectured invariant, setting `Y` to be 1. When `m` does this, it also ends up in a non-terminating loop. As a result, the broken invariant is never exposed to a call of `f` in the sequential 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 Section 3.1, we outline our programming model. In Section 3.2, we formalize the notions of executions and schedules. In Section 3.3, we review the notion of conflict-serializability and state some of its known properties.



```
module T;
 int Q=0, W=0, E=0;
 void t() {
   int q=0, w=0, e=0;
   acquire(Q);
   q = Q++;
   acquire(W);
   // assert(Q==W+1)
   release(Q);
   w = W++;
   // assert(q==w)
   if (isEven(q))
     acquire(E);
   release(W);
   if (isEven(w)) {
     e = E++;
     release(E);
   }
 }
```

(a) Module T

```
module DT;
 N R;

 bool delete(int k){
   N p = R;
   p.acquire();
   N c = p.r;
   bool ret = false;

   while (c != null){
     c.acquire();
     if (c.k == k)
       break;
     p.release();
     p = c;
     if (p.k < k)
       c = p.r;
     else
       c = p.l;
   }
                          if (c != null){
                            ret = true;
                            N cr = c.r;
                            c.r = null;
                            N cl = c.l;
                            c.l = null;
                            N t = cr;

                            if (cr == null)
                              t = cl;
                            else{
                              if (cl != null){
                                N rp = cr;
                                rp.acquire();
                                N rc = rp.l;
                                while (rc != null){
                                  rp.release();
                                  rp = rc;
                                                    rp.acquire();
                                                    rc = rp.l;
                                                  }
                                                  rp.l = cl;
                                                  rp.release();
                                                }
                                              }

                                              if (p.k < k)
                                                p.r=t;
                                              else
                                                p.l=t;
                                            }

                                            p.release();
                                            if (c != null)
                                              c.release();

                                            return ret;
                                          }
```

(b) Module DT

```
module M;
 int X=0, Y=0;
 void m() {
   acquire(X);
   if (*) {
     acquire(Y);
     Y = 1;
     release(Y);
     while (true)
       skip;
   }
   release(X);
 }
 void f() {
   int y;
   acquire(Y);
   // assert(Y==0)
   y = Y;
   release(Y);
   if (y!=0)
     acquire(Y);
   if (y!=0)
     release(Y);
 }
```

(c) Module M

Figure 1. Concurrent modules. `(*)` represents non-deterministic branching. `N` is the type of a pointer to the user defined type {`N l,r; int k`}. The `assert` statements are used for expository reasons only. They are not intended to be executable code, and thus remarked.

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, i.e.*, a lock can be held by at most one thread at a time. The execution of a thread trying to acquire a lock which is held by another thread is *blocked* until a time when the lock is *available, i.e.*, is not held by any thread.

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* $x \in \mathcal{V} = \mathcal{V}_L \uplus \mathcal{V}_G$ of variable identifiers, *local variable identifiers* $x, y \in \mathcal{V}_L$ and *global variable identifiers* $X, Y \in \mathcal{V}_G$, $f \in \mathcal{F}$ of field identifiers, $p \in \mathcal{P}$ 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 ourself 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*, shown in Figure 2, which have their expected behavior. 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 between these and local variables and `acquire/release` instructions. In particular, the condition of an `assume` statement cannot refer to a global variable.

$$
\begin{array}{rcl}
\mathit{stms} & = & \texttt{skip} \mid \texttt{x = e(}y_1\texttt{,...,}y_k\texttt{)} \mid \texttt{assume(b)} \\
& \mid & \texttt{x = Y} \mid \texttt{Y = x} \\
& \mid & \texttt{acquire Y} \mid \texttt{release Y} \mid \\
& \mid & \texttt{x.acquire()} \mid \texttt{x.release()} \\
& \mid & \texttt{x = new R()} \mid \texttt{x = y.f} \mid \texttt{x.f = y}
\end{array}
$$

Figure 2. The primitive instructions used in this paper. b stands for a local boolean variable. `e(`y_1`,...,`y_k`)` stands for an arbitrary expression over local variables.

3.1.2 Semantics

Memory states. Figure 3 defines the semantic domains of *memory states* of module m and the meta-variables ranging over them. We assume the semantic domain $t \in \mathcal{T}$ of thread identifiers.

A *memory state* $\sigma = \langle g, h, \varrho \rangle \in \Sigma$ of a concurrent module cm 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 $v \in \mathcal{VAL}$ can be either a location, an integer, a boolean value, or *null*. ϱ associates a thread t with its thread local state $\varrho(t)$. A *thread-local state* $s = \langle \kappa, \rho, L \rangle \in \mathcal{S}$ is a triplet: κ is the value of the thread's program counter, ρ records the values of its local variables, and L 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 $\overset{tr}{\rightsquigarrow} \subseteq \Sigma \times \mathcal{T} \times (\mathcal{K} \times \mathcal{K}) \times \Sigma \uplus \{\bot\}$ that interleaves the execution of different threads. ($\bot$ is a specially designated error state.) A *transition* $\sigma \xrightarrow{\langle t,e \rangle} \sigma' \in \overset{tr}{\rightsquigarrow}$ represents the fact that σ can be transformed into σ' via thread t executing the instruction annotating control-flow edge e. Given a transition $\sigma \xrightarrow{\langle t,e \rangle} \sigma'$, we say that the transition is *executed* by t.

$$
\begin{array}{rcrl}
v & \in & \mathit{Val} & = & \mathit{Loc} \uplus \mathcal{Z} \uplus \{tt, ff, null\} \\
\rho & \in & \mathcal{E} & = & \mathcal{V}_{\mathcal{L}} \hookrightarrow \mathit{Val} \\
h & \in & \mathcal{H} & = & \mathit{Loc} \hookrightarrow \mathcal{F} \hookrightarrow \mathit{Val} \\
l & \in & \mathcal{L} & = & \mathit{Loc} \uplus \mathcal{V}_{\mathcal{G}} \\
s & \in & \mathcal{S} & = & \mathcal{K} \times \mathcal{E} \times 2^{\mathcal{L}} \\
\sigma & \in & \Sigma & = & (\mathcal{V}_{\mathcal{G}} \hookrightarrow \mathit{Val}) \times \mathcal{H} \times (\mathcal{T} \hookrightarrow \mathcal{S})
\end{array}
$$

Figure 3. Semantic domains.

We present the (rather mundane) formal definition of transitions pertaining to a specific set of primitive instructions in [2]. The statement "`acquire Y`" attempts to acquire a lock on global variable `Y`. The statement "`x.acquire()`" attempts to acquire a lock on the heap object pointed to by local variable `x`. The `assume`(b) instruction acts as `skip` when executed on a state σ which satisfies b. If b does not hold in σ, 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 $\langle t, e\rangle$ consisting of a thread identifier and a control-flow edge as an *execution step*. We say that $\langle t, e\rangle$ is *enabled* in state σ if there exists some transition $\sigma \stackrel{\langle t,e\rangle}{\rightarrow} \sigma'$ and that it is *disabled* otherwise. Assume that the program-counter of t is at the source of edge e (in state σ). Then, $\langle t, e\rangle$ is disabled in σ iff 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* π is a sequence of transitions $\sigma_0 \stackrel{\tau_1}{\rightarrow} \sigma_1 \ldots \stackrel{\tau_k}{\rightarrow} \sigma_k$ that starts at the initial state (*i.e.*, $\sigma_0 = \sigma_\emptyset$) 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 π is the subsequence of transitions executed by thread t in π.

The *transaction* T_i executed by a thread t_i in an execution π is the sequence of control-flow edges annotating the transitions executed by t. Note that a transaction is a path in the control flow graph of the procedure p invoked by t, 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* $\Phi = \langle t_1, e_1\rangle, \ldots, \langle t_k, e_k\rangle$ is a sequence of execution steps. Φ 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. Φ is *feasible* if it is *induced* by some execution π, *i.e.*, $\pi = \sigma_0 \stackrel{\langle t_1,e_1\rangle}{\longrightarrow} \sigma_1 \ldots \stackrel{\langle t_k,e_k\rangle}{\longrightarrow} \sigma_k$.

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 π_1 and π_2 are *conflict-equivalent* if they include the same transactions and they agree on the order between conflicting transitions. (I.e., the ith transition of a thread t precedes and conflicts with the jth transition of thread t' in π_1 iff the former precedes and conflict with the latter in π_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 π and π' be conflict-equivalent (well-locked) executions. (a) For any thread t, the set of locks held by t after π is the same as the set of locks held by t after π'. (b) The execution of (t, e) is enabled after π iff it is enabled after π'.*

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

LEMMA 2. *Let $\pi_{ni} = \pi_P \pi_t \pi_S$ be a well-locked NI-execution, where π_t is the sub-execution by a thread t. If the execution step (t, e) is enabled after π_{ni}, then it is also enabled after $\pi_P \pi_t$.*

Proof: We first show that the set of locks free at the end of $\pi_P \pi_t$ is a superset of the set of locks free at the end of π_{ni}. Consider any lock ℓ held by some thread t' after the execution of $\pi_P \pi_t$. Since t' does not execute during π_S, it will continue to hold lock ℓ at the end of π_{ni}. So the set of locks available at the state produced by $\pi_P \pi_t$ is a superset of the set of available locks in the state produced by π_{ni}.

Note that an execution of (t, e) can be disabled only for two reasons: e represents a conditional branch (with an `assume` statement whose condition references only local variables) that evaluates to false or e tries to acquire a lock that is not free. The local state of thread t is the same after π_{ni} as after $\pi_P \pi_t$, so the same conditional branches will be enabled in both cases. Furthermore, any lock that can be acquired after π_{ni} can also be acquired after $\pi_P \pi_t$ (by the claim we proved). The lemma follows. ■

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 $\sigma = \langle g, h, \varrho \rangle$, the *thread-owned state* of thread t, defined by $\mathit{owned}_t(\sigma) = \langle g|_L, h|_L, \varrho(t) \rangle$ where $\varrho(t) = \langle \kappa, \rho, L \rangle$, is its own thread-local state in σ 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 π is obtained by replacing every state σ in the corresponding t's sub-execution by $\mathit{owned}_t(\sigma)$.

We now define a certain class of properties of executions. To avoid restricting ourself to any particular logic we simply identify a property ϕ (of executions) with the set of executions that satisfy the property. Similarly, we identify a property ϕ' of thread-owned views with the set of thread-owned views that satisfy the property.

We say that a property ϕ of executions is *transaction-local* if (a) there is a property ϕ' of thread-owned views such that an execution π is in ϕ iff all the thread-owned views of all the threads launched in π are in ϕ', and (b) ϕ is prefix-closed: if π is in ϕ, then every prefix of π is in ϕ. For any transaction-local property ϕ of executions, we will use ϕ_T to denote the property ϕ' 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) $\pi \in \text{LP}$ implies that π 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 π 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 π 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

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

Proof: By definition of a transaction-local property, there exists a property LP_T of thread-owned views such that an execution follows the locking protocol LP *if and only if* the thread-owned views it generates are in LP_T. (Essentially, LP_T checks if a particular thread in a given execution satisfies the locking protocol.)

Assume that both $\pi_P \pi_t \pi_S$ and $\pi_P \pi_t \xrightarrow{t,e} \sigma'$ satisfy LP. We need to show that $\pi_P \pi_t \pi_S \xrightarrow{t,e} \sigma$ also satisfies LP. The thread-owned view of t in $\pi_P \pi_t \pi_S \xrightarrow{t,e} \sigma$ is the same as the thread-owned view of t in $\pi_P \pi_t \xrightarrow{t,e} \sigma'$. The thread-owned view of any thread $t' \neq t$ in $\pi_P \pi_t \pi_S \xrightarrow{t,e} \sigma$ is the same as the thread-owned view of s in $\pi_P \pi_t \pi_S$. The result follows. ■

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 π. 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 σ' is the state produced by a thread t executing an instruction st annotating e. By the induction hypothesis, execution π follows the locking protocol. The locking protocol guarantees conflict-serializability. Thus, there exists a non-interleaved execution π_{ni} which is conflict-equivalent to π.

By Lemma 1, the execution of (t, e) is enabled after π_{ni} as well. Let π_{ni} be of the form $\pi_P \pi_t \pi_S$ where π_t is the sub-execution by thread t. By Lemma 2, (t, e) is also enabled after $\pi_P \pi_t$.

Since $\pi_P \pi_t.(t, e)$ is an NI-execution, it satisfies LP (by assumption). By Lemma 3, $\pi_{ni}.(t, e)$ satisfies LP as well. Since π satisfies LP, and π and π_{ni} are conflict-equivalent (and produce the same state), it follows that $\pi.(t, e)$ 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 Figure 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 *completable* 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 completable. (One way to show that every ACNI-execution is completable 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 Section 5.4.

5.1 Proof Strategy

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

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

The next lemma is a simple consequence of the definitions.

LEMMA 5. *Let π_{cni} be an equivalent completion of π_{ni}. For any thread t, the thread-owned view of t in π_{ni} is a prefix of the thread-owned view of t in π_{cni}.*

Our overall proof strategy is to take an NI-execution π_{ni} and construct an equivalent completion π_{cni}. By Lemma 5, if π_{cni} satisfies LP, then π_{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 π_{ni}.

Let the schedule of π_{ni} be $\alpha_1 \cdots \alpha_k$, each α_i representing the sub-schedule of a different thread t_i. We construct a sequence of schedules γ_h, where $1 \leq h \leq k$, where each γ_h is of the form $\alpha_1\beta_1 \cdots \alpha_h\beta_h$ and is the schedule of an CNI-execution. Furthermore, the execution of β_i is guaranteed to not access any variable accessed by any α_j for $j > i$.

We let γ_0 be the empty schedule. We construct the schedule γ_{i+1} from γ_i as follows. Note that $\gamma_i\alpha_{i+1}$ is the schedule of an ACNI-execution. Hence, by assumption, this is completable. That is, there exists an CNI-execution π_{i+1} with a schedule $\gamma_i\alpha_{i+1}\beta_{i+1}$.

If the execution of β_{i+1} in π_{i+1} does not access any variable accessed by α_j in π_{ni} for any $j > i+1$, then we define γ_{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 π_k produced finally to be the *iterative completion* of π_{ni}. Otherwise, we say that the *iterative completion* of π_{ni} is not defined.

LEMMA 6. *If the iterative completion of an NI-execution π_{ni} is defined, then it is an equivalent completion of π_{ni}.*

It follows from the above discussion that the key open issue in proving ACNI-reduction is handling the case where β_{i+1} accesses some variable accessed by some α_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 Section 5.4).

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

LEMMA 7. *Assume that every ACNI-execution of a module satisfies an LCS-locking-protocol* LP. *Let π be an NI-execution that has an equivalent completion. Then, any NI-execution $\pi' = \pi\tau$ consisting of π following by a single transition τ satisfies* LP.

Proof: Let transition τ be generated by thread t executing instruction e. Since π satisfies LP, the sub-execution of every thread in π satisfies LP. Thus, to prove that π' satisfies LP we just need to verify that the sub-execution of t in π' satisfies LP.

Let α denote the sub-execution of t in π. Thus, the sub-execution of t in π' is $\alpha\tau$. Let π_{cni} be an equivalent completion of π, which exists by assumption. Consider the ACNI-execution π^t which is the prefix of π_{cni} that ends with α. (We let π^t be π_{cni} if α 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 π. Since t can execute instruction e after π, it follows that it can execute the same instruction after π^t as well. (See Lemma 1.) The sub-execution of t in π^t extended by e must satisfy LP (since this extension is an ACNI-execution). Hence, $\alpha\tau$ must also satisfy LP.

It follows that π' satisfies LP. ■

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 completable, then every NI-execution π of the module (a) satisfies* TL *and (b) has an equivalent completion π_{cni}.*

Proof: Assume that every ACNI-execution of the module satisfies TL and is completable. 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 π satisfies (a) and (b). Consider an NI-execution $\pi' = \pi \stackrel{\langle t,e \rangle}{\longrightarrow} \sigma'$, where σ' is the state produced by the execution of an instruction e by a thread t after π.

(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 π' may not be defined, as illustrated by the following simple example. Assume that π' 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 π' is not defined. The trick in this case is to reorder the execution of t_1 and t_2 in π' to obtain a conflict-equivalent execution π'' and construct the iterative completion of π''. We now show that this can always be done.

Let the schedule of π' be $\alpha_1 \cdots \alpha_k$, each α_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 π'. (A thread t_i that does not acquire any locks can be easily handled.) Consider the state σ_i produced by the execution π_i of $\alpha_1 \cdots \alpha_i$. Let $locked_i$ denote the set of variables locked by t_i in σ_i. Let $lockable_i$ denote the set of children of variables in $locked_i$ that have not already been locked by t_i (in the execution π_i). (These are variables that t_i can immediately lock, if available, without violating TL.) Let $lockable_i^*$ denote the set of descendants of variables in $lockable_i$ (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_i *may-depend-upon* t_j if the first variable locked by t_j (in π') is contained in $lockable_i^*$. These are the dependences that can create us problem while constructing an iterative completion.

We identify a permutation of the set of threads t_1 to t_k in which t_i occurs after t_j if t_i 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_1 through t_j, we identify S_{j+1} as follows. If none of the t_i in S_j may-depend-upon t_{j+1}, we append t_{j+1} 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_{i+1}. Let the final sequence S_k be $t_{p_1}, \cdots, t_{p_k}$.

Consider the schedule $\alpha_{p_1} \cdots \alpha_{p_k}$. We claim that this produces an NI-execution π'' that is conflict-equivalent to π'. Furthermore, we claim that the iterative completion of π'' is defined and that this yields an equivalent completion for π' 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 π, we say that t_2 *depends* on t_1 if t_1 accesses a global variable g during the execution that is accessed later by thread t_2. We define the *schedule slice* of π with respect to thread t, to be the subsequence of the schedule of π consisting of steps taken by t or any other thread t' on which t depends (directly or transitively). It follows trivially that the schedule slice is a feasible schedule. We define the *slice* of π with respect to t, denoted $\pi|_t$, to be the execution induced by the schedule slice. For any thread t' that occurs in $\pi|_t$, it can be shown that the thread-owned view is the same in π and $\pi|_t$.

THEOREM 9. *If every ACNI-execution of a module satisfies 2PL and is completable, then (a) every NI-execution π of the module satisfies 2PL, and (b) for every NI-execution π of the module and every thread t, $\pi|_t$ has an equivalent completion π^t_{cni}.*

Proof: The proof is by induction on the length of the execution. Consider an NI-execution $\pi' = \pi \stackrel{\langle t,e\rangle}{\rightarrow} \sigma'$, where we assume that π satisfies (a) and (b). We show that π' also satisfies (a) and (b).

(a) Since π follows 2PL, we just need to show that t follows 2PL in π'. Consider $\pi'|_t$ which must be of the form $\pi|_t \stackrel{t,e}{\rightarrow} \sigma''$. It follows from Lemma 7 and the inductive hypothesis that $\pi'|_t$ follows 2PL. Therefore, π' itself satisfies (a) (*i.e.*, follows 2PL).

(b) We now argue that the iterative completion of $\pi'|_t$ is defined. Assume that we take a schedule $\alpha_1 \cdots \alpha_k$ and construct the schedule $\alpha_1\beta_1 \cdots \alpha_k\beta_k$ using our iterative completion algorithm. We just need to show that the set of variables accessed by β_i is disjoint from the set of variables accessed by α_j for $j > i$. Note that the execution of the thread t_i ($i < k$) is included in $\pi'|_t$ only if some other thread depends on t_i in π'. Such a dependence is possible only if thread t_i has reached its shrinking phase and has released some lock by the end of α_i. In this case, β_i cannot acquire any new lock. Hence, β_i can access only variables locked by t_i at the end of α_i. But α_j can only access variables that are *not* locked by t_i at the end of α_i. The result follows. ■

5.4 ACNI-Reduction for Progressive Locking Protocols

We now generalize the results presented above to show that ACNI-reduction is valid for a class of locking protocols.

A thread (or a transaction) in an execution is *visible* if it has released at least one lock. In particular, if a thread t reads/writes a global variable that is then subsequently accessed in a well-locked execution by another thread, then the first thread t is visible.

An LCS-locking-protocol LP is said to be *progressive* if (i) An execution satisfying LP cannot contain a deadlock involving a visible transaction, and (ii) If an execution π satisfies LP and t is any thread in π, then the *abrupt completion* of t in π, defined to consist of π followed by a release by t of all locks it holds, also satisfies LP. Note that both TL and 2PL satisfy the above definition and that condition (ii) above is satisfied by any locking protocol that allows threads to release a lock at any time.

The following theorem (whose proof appears in Appendix A) states that ACNI-reduction is valid for any progressive LCS-locking-protocol.

THEOREM 10. *Let LP be a progressive locking protocol. If every ACNI-execution of a module satisfies LP and is completable, then (a) every NI-execution π of the module satisfies LP, and (b) for every NI-execution π of the module and every thread t, $\pi|_t$ has an equivalent completion π^t_{cni}.*

6. Sequential Reductions for Verifying DTL

In this section we extend our results to hand-over-hand locking protocols, such as the concurrent binary tree example shown in Figure 1(b). This requires us to generalize our results to handle mutable, dynamically allocated, pointer-linked data structures with heap-storable locks. More importantly, however, the protocol used is a *Dynamic Tree Locking* protocol. The tree-ordering, which determines the order in which a thread should acquire locks, is determined by the data-structure itself and can change over time. This requires extensions to the protocol itself. Significantly, the required extension makes the protocol a *non-transaction-local* property, requiring new proofs of the reduction theorems.

6.1 Dynamic Tree Locking (DTL)

In this section, we present DTL, a variant of the Dynamic Dag Locking protocol introduced in [7]. The variations permit coding patterns (such as temporary violations of structural invariants in the part of the graph "locked" by a thread) typically found in implementations. DTL guarantees conflict-serializability, though we omit a proof of this due to space constraints.[1]

We simplify the definition of the dynamic tree locking protocol, the statement of the theorems and their proofs by 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 $u \rightarrow v$ iff the variable/object represented by u 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 $u \stackrel{f}{\rightarrow} v$. 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 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 π_1 and π_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 π_1 *corresponds* to the object allocated by the i-th execution-step of thread t in π_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 $u \rightarrow v$ only if v is in the thread's scope. (More precisely, the thread can assign the address of*

[1] The reduction theorems for the Dynamic Dag Locking protocol can be found in [2].

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, u 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 u 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 u 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 π and π' be the same. Let the execution of step (t, e) be enabled after both π and π'. Then, the execution of step (t, e) after π satisfies* DTL *conditions 1 to 5 iff the execution of step (t, e) after π 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 $\pi_{ni} = \pi_P \pi_t \pi_S$ be a non-interleaved execution, where π_t is the sub-execution of a thread t. Let $\pi_{ni} \stackrel{t,e}{\rightarrow} \sigma$ be an execution that extends π_{ni} by a single transition. If $\pi_P \pi_t \pi_S$ satisfies* DTL *and $\pi_P \pi_t \stackrel{t,e}{\rightarrow} \sigma'$ satisfies* DTL*, then $\pi_P \pi_t \pi_S \stackrel{t,e}{\rightarrow} \sigma$ also satisfies* DTL*.*

Proof: Note that the thread-owned view of t is the same in $\pi_P \pi_t$ and $\pi_P \pi_t \pi_S$. Since the step (t, e) satisfies DTL after $\pi_P \pi_t$, 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 $\pi_P \pi_t \pi_S$. Thus, v must be in the scope of t throughout the execution of π_S. Consequently, v must have the same set of predecessors after $\pi_P \pi_t \pi_S$ as after $\pi_P \pi_t$. Thus condition 6 is satisfied by $\pi_P \pi_t \pi_S \stackrel{t,e}{\rightarrow} \sigma$ iff it is satisfied by $\pi_P \pi_t \stackrel{t,e}{\rightarrow} \sigma'$. ■

LEMMA 14. *Let $\alpha \beta_u \gamma \delta_t$ denote the schedule of an NI-execution, where β_u and δ_t denote the schedules executed by threads u and t respectively. Assume that no transitive conflict-dependence exists between u and t in this execution. If $\alpha \beta_u \gamma$ and $\alpha \gamma \delta_t$ both satisfy* DTL*, then so does $\alpha \beta_u \gamma \delta_t$. (In other words, β_u does not affect the correctness of δ_t.)*

Proof: Let π_1 denote the execution of $\alpha \beta_u \gamma \delta_t$ We need to show that the execution of all instructions in δ_t in π_1 satisfy DTL. Let π_2 denote the execution of $\alpha \gamma \delta_t$. The absence of any transitive conflict-dependence between β_u and δ_t in π_1 means that Lemma 12 applies and that we just need to verify that the execution of δ_t in π_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 δ_t and their successors at any point during the execution of δ_t in π_1 is isomorphic to the *corresponding* subgraph induced in π_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 π_1 can have more than one predecessor iff the corresponding vertex going out of t's scope at the corresponding point in π_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 $\alpha_1 \alpha_2 \cdots \alpha_k$ be the schedule of an* DTL *NI-execution π, each α_i represents the schedule of a different thread t_i. Let $\alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k \beta_k$ be the schedule another* DTL *NI-execution π', where $\alpha_i \beta_i$ represents the schedule of thread t_i. Assume that t_i acquires at least a single lock in execution π. For any $i < j$, the set of locations accessed (read, written, or locked) during the execution of β_i in π' is disjoint from the set of locations accessed during the execution of α_j in π. (Note that we compare heap locations in different executions modulo the correspondence relation described in Sec. 6.1.)*

Proof: Let σ_i be the state produced by the execution of $\alpha_1 \cdots \alpha_i$. Let V_i denote the set of vertices in σ_i. Let $scope_i$ denote the set of vertices in the scope of t_i in state σ_i. Let $reachable_i$ denote the set of vertices in σ_i that are reachable via some path from some vertex in $scope_i$.

For $j > i$, the execution of α_j in π *cannot* lock any vertex in $reachable_i$. We can establish this inductively. Thread t_j cannot lock any vertex in the scope of any other thread t_k in σ_i. Furthermore, any vertex y not in the scope of any thread (in σ_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 $\alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k \beta_k$. Let us identify any vertex allocated in a step s in π with the location allocated by the corresponding step s' in π'. We claim that any vertex in V_i that is locked by t_i during the execution of β_i must be in $reachable_i$: 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 β_i and α_j must be disjoint (where $i < j$). ■

THEOREM 17. *If every ACNI-execution of a module satisfies* DTL *and is completable, then (a) every NI-execution π of the module satisfies* DTL*, and (b) every NI-execution π of the module has an equivalent completion π_{cni} (which is a CNI-execution, by definition).*

Proof: Assume that every ACNI-execution of the module satisfies TL and is completable. 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 π satisfies (a) and

(b). Consider an NI-execution $\pi' = \pi \stackrel{\langle t,e \rangle}{\rightarrow} \sigma$, where σ is the state produced by the execution of an instruction e by a thread t after π. We assume that t has a transitive conflict-dependence on all other threads that execute in π. (Otherwise, we can omit the execution of any other thread u that t does not have a dependence on from π' to get a shorter execution π''. From the inductive assumption, π'' must satisfy DTL. By Lemma 14, π' must also satisfy DTL.)

Let π_{cni} be an equivalent completion of π. π_{cni} must satisfy DTL, by assumption, since it is an ACNI-execution. Let the schedule of π be $\alpha_1\alpha_2\cdots\alpha_k$, and the schedule of π_{cni} be $\alpha_1\beta_1\alpha_2\beta_2\cdots\alpha_k\beta_k$, where each α_i (and β_i) represents execution by a thread t_i. Note that Lemma 16 applies to π and π_{cni}.

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 π_{cni} as well. Suppose that (t, e) tries to acquire a lock on u. Since (t, e) is enabled after π, u must be unlocked after π. 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 α_k in π). The definition of an equivalent completion implies that t_k acquires and releases a lock on u during α_k in π_{cni}. Hence, t_k cannot acquire a lock on u again during β_k. Hence, u must be unlocked after π_{cni} as well. Thus, (t, e) is enabled after π_{cni}.

Now $\pi_{cni} \stackrel{t,e}{\rightarrow} \sigma'$ is an ACNI-execution, which satisfies DTL by assumption. So, the execution of (t, e) following π_{cni} satisfies DTL. As a result, the execution of (t, e) after π 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 $\pi_{cni} \stackrel{t,e}{\rightarrow} \sigma'$ would violate DTL.) Hence, (a) follows.

As for (b), the ACNI-execution $\pi_{cni} \stackrel{t,e}{\rightarrow} \sigma'$ must have a completion by assumption. This gives us an equivalent completion for π'. (Lemma 16 above ensures that this is an equivalent completion.)

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

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 π'. (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 ϕ be a transaction-local property. If every NI-execution of module M is well-locked and satisfies property ϕ, then every execution of the module satisfies ϕ.*

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

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

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 Figure 1(a). The property `Q == 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 == 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 σ is *quiescent* if there exists a complete execution that produces σ. A property ϕ of program states is said to be a *module invariant* if all quiescent states satisfy ϕ. Our results also justify the use of sequential analyses to infer module invariants. For example, a sequential analysis of module `T`, shown in Figure 1(a) can determine that `Q == 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 Figure 4(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.

Completability 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 Figure 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` in Figure 1(a) and cannot verify that the module follows the TL policy and is, hence, conflict-serializable. However, both an optimistic analysis or a cautiously optimistic analysis, described below, using, e.g., the octagon domain [22], can verify these properties.) Such an analysis is typically insufficient for verifying adherence to DTL since DTL depends on module shape invariants. Verification using type systems are often pessimistic analyses and, when they are, can safely be used for conflict-serializable modules.

Cautiously Optimistic Analysis. This middle ground between the two possibilities mentioned above is similar to the optimistic analysis in computing a module invariant, except that it computes a weaker module invariant, exploiting only the NI-reduction. Specifically, the module invariant computed by this approach describes all states that can be produced by any NI-execution. The condition for applying this approach is the same as for the pessimistic approach, namely conflict-serializability.

Note. A key advantage of sequential reduction is that it permits analyses to track correlations between variables, *including those not locked*, without having to account for the side-effect of other threads on such correlations.

```
module MGC-DTL;
  void mostGeneralClient() {
    DTL t;
    t.init();
    while (*) {
      int k = random();

      if (*)
        t.insert(k);
      else if(*)
        t.delete(k);
      else
        t.contains(k);
    }
}
```

(a) The MSGC of module DT.

```
module S;
  int Y, B;
  void p() {
    acquire(B);
    B = 0;
    release(B);
    acquire(B);
    int b = B;
    if (b) Y = 2;
    release(B);
  }
  void q() {
    acquire(B);
    B = 1;
    release(B);
  }
```

(b) Module S.

Figure 4. Additional examples.

8. Prototype Implementation

To measure the value of sequential reduction, we analyzed the most general *concurrent* client of a lock-coupling sorted list implementation with different numbers of threads. The analysis, an adaptation of [1] to verify memory safety, tracks properties such as pointed-to by a variable, reachability from a variable, heap sharing, cyclicity, and sortedness. Analysis with one thread, corresponding to optimistic analysis obtained by our sequential reduction results, took 10 seconds and consumed 3.6 MB memory. obtained by a sequential reduction.) With two threads, the analysis took almost 4 hours and consumed 886 MB memory. With three threads, the analysis did not terminate in 8 hours.

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 DTL 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 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.) Table 1(b) compares the cost of our analysis ($\star$) 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.

Table 1(a) compares the cost of our analysis ($\star$) to existing analyses using the lock-coupling list algorithm as a common benchmark. The analyses of [1], [24], and [27] verify linearizability [14]. Our analysis verifies serializability. These problems are similar, but not the same. Nevertheless, we believe the comparison shows the effectiveness of sequential reasoning. The analysis of [1] is able to verify that the list coupling algorithm is linearizable for a bounded number of threads. In our experiments, the analysis timed

	(a) LC-List				(b) LC-Tree	
	$\star$	[1]	[24]	[27]	$\star$	Vanilla
Time (sec.)	3.5	T/O	596.1	0.45	124.6	58.4
Space (MB)	4.0	T/O	90.3	0.23	90.6	43.9

Table 1. Experimental Results. Experiments performed on a machine with a 2.4 Ghz Intel Core 2 Duo processor and 2 Gb memory running version 10.5.7 of the Mac OS X operating system.

out. However, in [1] it is reported to verify the algorithm for two threads in less than 4 hours. The analysis of [24], which is an optimized version of [4, 21], is 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.

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 incomparable with theirs and complement their work: The applicability conditions for their result do not hold for the problem we study. For example, the code fragment that can be *reduced* (i.e., treated atomically) by their approach cannot span blocking synchronization operations (such as an acquire-lock operation). In contrast, we are able to treat code fragments containing an arbitrary number of such operations atomically, when they satisfy the locking protocol. On the other hand, their technique, unlike ours, is not limited to lock-based synchronization.

One of the early works on reductions is Lipton's theory of D-reductions which permits treating certain blocks of instructions atomically [19]. Lipton's theory discusses the conditions that the instructions in the block need to satisfy (certain commutativity properties of different instructions in every *interleaved* execution) to enable such reduction. Lipton does not discuss the problem 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 Figure 4(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, Mooly 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

[1] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.

[2] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential verification of serializability. Technical report. http://www.dcs.qmul.ac.uk/∼maon/pubs/seqser.pdf.

[3] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[4] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, 2008.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, 2005.

[7] V. K. Chaudhri and V. Hadzilacos. Safe locking policies for dynamic databases. In *PODS*, 1995.

[8] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, 2008.

[9] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, 2009.

[12] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *JACM*, 45(5), 1998.

[13] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[15] Z. M. Kedem and A. Silberschatz. A characterization of database graphs admitting a simple locking protocol. *Acta Informatica*, 16, 1981.

[16] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.

[17] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA*, 2000.

[18] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS*. Springer, 2000.

[19] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *CACM*, 18(12), 1975.

[20] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In *SAS*, 2006.

[21] R. Manevich, T. Lev-Ami, G. Ramalingam, M. Sagiv, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS*, 2008.

[22] A. Miné. The octagon abstract domain. *HOSC*, 19(1), 2006.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.

[24] M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, 2009.

[25] S. D. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. *FMSD*, 28(3), 2006.

[26] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2), 1979.

[27] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, 2009.

[28] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufmann, 2001.

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 completable. Let π_{ni} be any non-interleaved execution that satisfies* LP. *If all threads except the one executing last in π_{ni} are visible then π_{ni} has an equivalent completion.*

Proof: Let the schedule of the given NI-execution π_{ni} be of the form $\alpha_1\alpha_2\cdots\alpha_k$, where each α_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 β_i do not modify (lock) any global variable that is read by α_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 π_{ni} 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 Φ_0, and construct the schedule Φ_{i+1} from Φ_i by identifying the thread t_{s_i} to be scheduled next, constructing a thread-completion $\gamma_{s_i} = \alpha_{s_i}\beta_{s_i}$, and defining $\Phi_{i+1} = \Phi_i\gamma_{s_i}$. Let S_i denote the set $\{s_1, \cdots, s_i\}$. We maintain the following invariant as we construct the schedule: Φ_i is a feasible schedule; Φ_i preserves dependencies among the scheduled threads (*i.e.*, if we have thread t_{s_q} is dependent on thread t_{s_p} in π_{ni}, then we will ensure that $p < q$); further, in the execution of Φ_i, the instructions executed by β_{s_i} will not reference (or lock) any global variable referenced by α_j for any j not in S_i (*i.e.*, any thread that has not been scheduled yet).

Given schedule Φ_i, we say that a thread t_j has been completed if $j \in S_i$. We say that a thread t_j is a *candidate* (for scheduling next) if all threads it depends on (in execution π_{ni}) 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 π_{ni}. 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 β_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 Φ_{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 t_j has a potential completion $\alpha_j\beta_j$. Since the candidate is not valid, β_j must have a prefix of the form $\beta'_j e_j$, where β'_j does not reference any "forbidden variable" (any variable referenced by a thread not in $S_i \cup \{t_j\}$), while e_j references a forbidden variable. Assume that e_j references a forbidden variable also referenced by α_h, where $t_h \notin S_i \cup \{t_j\}$. We say that t_j is *blocked* by t_h, and say there is a *blocking dependence* from t_h 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 δ_j denote $\alpha_j\beta'_j$. Since $\alpha_j\beta_j$ satisfies LP (as mentioned above), it must be well-locked and hence e_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 α_j; thus, the forbidden variables cannot include any variable for which t_j holds the lock at the end of α_j.)

Let $t_{c_1}, \cdots, t_{c_x}$ be the set of candidate threads. Let $t_{d_1}, \cdots, t_{d_y}$ be the set of remaining unscheduled threads in an order satisfying the dependencies between them. Consider the following schedule $\Phi' = \Phi_i\delta_{c_1}\cdots\delta_{c_x}\alpha_{d_1}\cdots\alpha_{d_y}$. This is a feasible schedule that satisfies LP. Let Φ'' denote the abrupt completion of this schedule with respect to every t_{d_p}. 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 t_{c_q} that is trying acquire a (forbidden) lock that is not currently held by another candidate. We let t_{c_q} 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 π. The claim is trivial for the base case, when π is empty. Consider the inductive step. Part (a) follows from our inductive hypothesis and Lemma 7. From part (a), it follows that π and, hence, $\pi|_t$ are conflict-serializable. Let π_{ni} be the NI-execution obtained by conflict-serialization of $\pi|_t$. Part (b) follows by applying Lemma 20. ■

B. Sequential Reductions for Verifying Transaction-Local Properties

B.1 NI-Reduction For Transaction-Local Properties

LEMMA 21. *Let π be a conflict-serializable execution. Let π' be an execution that is conflict-equivalent to π. The following holds: (i) π is well-locked iff π' is well-locked, (ii) If π is well-locked, then π and π' generate the same thread-owned views.*

Proof: (i) Executions π and π' are conflict equivalent, thus in both executions the same threads are launched to execute the same transactions. If π is well-locked, every time thread t executes an instruction st which accesses a global integer variable, it holds its lock. Thus, when t executes the corresponding instruction st in π', it must also hold the lock which protects it as thread t executes the same sequence of `acquire` and `release` instructions before executing st. Similarly, the converse holds.

(ii) Following the same reasoning as above, when thread t executes the kth instruction in π and in π' it holds the same locks. Because the executions are well-locked, only thread t can access any variable protected by the locks it holds. Furthermore, because both executions have the same order of conflicting operations, the values t reads from global variables by corresponding instructions

(and thus the values which it writes to them) are the same. Thus, a simple induction on the number of instruction that t executes shows that required result, where the key observation is that every access to a global integer variable is preceded by acquiring its protecting lock. ■

Proof: (Theorem 18) Consider any execution π. Since the module is conflict-serializable, there exists an NI-execution π_{ni} that is conflict equivalent to π. It follows from the definition of a transaction-local property that π satisfies ϕ iff every thread-owned view v of π satisfies ϕ_T. But it follows from Lemma 21 that π and π_{ni} generate the same thread-owned views. Since π_{ni} satisfies ϕ, π must also satisfy ϕ. ■

B.2 ACNI-Reduction for Transaction-Local Properties

We now present the proof of Theorem 19.

Proof: Let π be any execution of M. By the definition of a transaction-local property, π satisfies ϕ iff every thread-owned view in π satisfies ϕ_T. It follows from our assumptions and Lemma 20 that every thread-owned view v in π is also a thread-owned view of an ACNI-execution π_{cni}. Since every ACNI-execution of M satisfies property ϕ, the result follows. ■

C. Sequential Reductions for Verifying DDL

C.1 Dynamic Dag Locking (DDL)

DEFINITION 3. *We say that an execution satisfies the Dynamic Dag Locking Protocol (*DDL*) if it satisfies the following conditions:*

1. *The execution is well-locked.*
2. *A transaction never acquires a lock on an object u after it has released a lock on that object u.*
3. *A transaction can delete an edge (u, v) only when it holds a lock on both u and v.*
4. *A transaction can insert an edge (u, v) only when it holds a lock on both u and v.*
5. *A transaction can insert an edge (u, v) only if v is not the first object it locks.*
6. *The first lock a transaction acquires is on a root of the dag.*
7. *Subsequent to acquiring its first lock, a transaction can acquire a lock on u only if holds a lock on all the predecessors of u (at the time the lock is acquired). Furthermore, u is required to be a non-root, unless u is a new resource created by the transaction.*

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

C.2 Simple Properties of Well-Locked Executions

LEMMA 22. *Let π and π' be conflict-equivalent (well-locked) executions. (a) For any thread t, the set of locks held by t after π is the same as the set of locks held by t after π'. (b) The execution of (t, e) is enabled after π iff it is enabled after π'.*

Proof: Trivial since conflict-equivalent executions produce the same state. ■

LEMMA 23. *Let $\pi_{ni} = \pi_P \pi_t \pi_S$ be a well-locked NI-execution, where π_t is the sub-execution by a thread t. (a) The set of locks free at the end of $\pi_P \pi_t$ is a superset of the set of locks free at the end of π_{ni}. (b) If the execution step (t, e) is enabled after π_{ni}, then it is also enabled after $\pi_P \pi_t$.*

Proof: (a) Consider any lock ℓ held by some thread t' after the execution of $\pi_P \pi_t$. Since t' does not execute during π_S, it will continue to hold lock ℓ at the end of π_{ni}. So the set of locks available at the state produced by $\pi_P \pi_t$ is a superset of the set of available locks in the state produced by π_{ni}. (b) Note that an execution of (t, e) can be disabled only for two reasons: e represents a conditional branch (with an `assume` statement whose condition references only local variables) that evaluates to false or e tries to acquire a lock that is not free. The local state of thread t is the same after π_{ni} as after $\pi_P \pi_t$, so the same conditional branches will be enabled in both cases. Furthermore, any lock that can be acquired after π_{ni} can also be acquired after $\pi_P \pi_t$ from (a). The result follows. ■

C.3 Simple Properties of DDL

LEMMA 24. *Let the thread-owned views of thread t in two executions π and π' be the same. Let the execution of step (t, e) be enabled after both π and π'. Then, the execution of step (t, e) after π satisfies* DDL *conditions 1 to 5 iff the execution of step (t, e) after π satisfies* DDL *conditions 1 to 5*

Proof: Follows since these conditions are all transaction-local properties. ■

LEMMA 25. *Let $\pi_{ni} = \pi_P \pi_t \pi_S$ be a non-interleaved execution, where π_t is the sub-execution of a thread t. Let $\pi_{ni} \xrightarrow{t,e} \sigma$ be an execution that extends π_{ni} by a single transition. If $\pi_P \pi_t \pi_S$ satisfies* DDL *and $\pi_P \pi_t \xrightarrow{t,e} \sigma'$ satisfies* DDL*, then $\pi_P \pi_t \pi_S \xrightarrow{t,e} \sigma$ also satisfies* DDL.

Proof: Note that the thread-owned view of t is the same in $\pi_P \pi_t$ and $\pi_P \pi_t \pi_S$. Since the step (t, e) satisfies DDL after $\pi_P \pi_t$, it follows from Lemma 24 we just need to check for conditions 6-7.

First, consider the case when step (t, e) acquires a lock on a root r after $\pi_P \pi_t$. We need to show that r will be a root after $\pi_P \pi_t \pi_S$ as well. This follows since the execution of π_S cannot convert root r into a non-root vertex by condition 5. Now, consider the case when step (t, e) acquires a lock on a non-root u after $\pi_P \pi_t$. It follows that t holds a lock on all predecessors of u after $\pi_P \pi_t$. We need to show that this condition continues to be true after the execution of π_S. I.e., we need to show that the execution of π_S will not create any new predecessor for u. This follows directly from condition 4 and 7: no other thread can acquire a lock on u and, hence, insert any edge with target u. ■

LEMMA 26. *Let $\alpha \beta_u \gamma \delta_t$ denote the schedule of an NI-execution, where β_u and δ_t denote the schedules executed by threads u and t respectively. Assume that no transitive conflict-dependence exists between u and t in this execution. If $\alpha \beta_u \gamma$ and $\alpha \gamma \delta_t$ both satisfy* DDL*, then so does $\alpha \beta_u \gamma \delta_t$. (In other words, β_u does not affect the correctness of δ_t.*

Proof: Let π_1 denote the execution of $\alpha \beta_u \gamma \delta_t$ We need to show that the execution of all instructions in δ_t in π_1 satisfy DDL. Let π_2 denote the execution of $\alpha \gamma \delta_t$. The absence of any transitive conflict-dependence between β_u and δ_t in π_1 means that the execution of δ_t in π_1 mimics the execution of δ_t in π_2, which is assumed to satisfy DDL. It follows from Lemma 24 that we just need to check for conditions 6-7. These follow directly since β_u does not lock any vertex locked by δ_t: hence, the execution of β_u cannot affect the set of predecessors of any vertex locked by δ. ■

C.4 NI Reduction for DDL

THEOREM 27. *If every NI-execution of a module satisfies* DDL *then every execution of the module satisfies* DDL.

Proof: The proof is by induction on the length of π. 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 σ' is the state produced by a thread t executing an instruction st annotating

e. By the induction hypothesis, execution π follows the locking protocol. The locking protocol guarantees conflict-serializability. Thus, there exists a non-interleaved execution π_{ni} which is conflict-equivalent to π.

It follows from Lemma 22 that the execution of (t, e) is enabled after π_{ni} as well. Let π_{ni} be of the form $\pi_P \pi_t \pi_S$ where π_t is the sub-execution by thread t. It follows from Lemma 23 that the execution of (t, e) is enabled after $\pi_P \pi_t$ as well.

Since $\pi_P \pi_t.(t, e)$ is an NI-execution, it satisfies DDL (by assumption). It follows from Lemma 25 that $\pi_{ni}.(t, e)$ satisfies DDL as well. Since π satisfies DDL, and π and π_{ni} are conflict-equivalent (and produce the same state), it follows that $\pi.(t, e)$ satisfies DDL as well. ■

C.5 ACNI Reduction for DDL

LEMMA 28. *Let $\alpha_1 \alpha_2 \cdots \alpha_k$ be the schedule of an* DDL *NI-execution π, each α_i represents the schedule of a different thread t_i. Let $\alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k \beta_k$ be the schedule another* DDL *NI-execution π', where $\alpha_i \beta_i$ represents the schedule of thread t_i. Assume that t_i acquires at least a single lock in execution π. For any $i < j$, the set of locations accessed (read, written, or locked) during the execution of β_i in π' is disjoint from the set of locations accessed (read, written, or locked) during the execution of α_j in π. (Note that every execution step s in π has a corresponding execution step s' in π'. We identify any location allocated in a step s in π with the location allocated by the corresponding step s' in π' in comparing the two executions.)*

Proof: Let σ_i be the state produced by the execution of $\alpha_1 \cdots \alpha_i$. Let V_i denote the set of vertices in σ_i. Let $locked_i$ denote the set of vertices locked by t_i in state σ_i. Let $reachable_i$ denote the set of vertices in σ_i that are reachable via some path from some vertex in $locked_i$.

For $j > i$, the execution of α_j in π *cannot* lock any vertex in $reachable_i$. We can establish this inductively. Thread t_j cannot lock any vertex in $locked_i$ (since t_i holds the lock on such vertices). Furthermore, if we have an edge $x \rightarrow y$ in σ_i, and if no t_j ($j > i$) can acquire a lock on x, then the edge $x \rightarrow y$ cannot be deleted by such t_j. Hence, t_j cannot acquire a lock on y (from the DDL conditions for acquiring locks).

Now consider the execution of $\alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k \beta_k$. Let us identify any vertex allocated in a step s in π with the location allocated by the corresponding step s' in π'. We claim that any vertex in V_i that is locked by t_i during the execution of β_i must be in $reachable_i$: 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 β_i and α_j must be disjoint (where $i < j$). ■

THEOREM 29. *If every ACNI-execution of a module satisfies* DDL *and is completable, then (a) every NI-execution π of the module satisfies* DDL*, and (b) every NI-execution π of the module has an equivalent completion π_{cni} (which is a CNI-execution, by definition).*

Proof: Assume that every ACNI-execution of the module satisfies TL and is completable. 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 π satisfies (a) and (b). Consider an NI-execution $\pi' = \pi \stackrel{\langle t,e \rangle}{\rightarrow} \sigma$, where σ is the state produced by the execution of an instruction e by a thread t after π. We assume that t has a transitive conflict-dependence on all other threads that execute in π. (Otherwise, we can omit the execution of any other thread u that t does not have a dependence on from π' to get a shorter execution π''. From the inductive assumption, π'' must satisfy DDL. It follows from Lemma 26 that π' must also satisfy DDL.)

Let π_{cni} be an equivalent completion of π. π_{cni} must satisfy DDL, by assumption, since it is an ACNI-execution. Let the schedule of π be $\alpha_1 \alpha_2 \cdots \alpha_k$, and the schedule of π_{cni} be $\alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k \beta_k$, where each α_i (and β_i) represents execution by a thread t_i. Note that Lemma 28 applies to π and π_{cni}.

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 π_{cni} as well. Suppose that (t, e) tries to acquire a lock on u. Since (t, e) is enabled after π, u must be unlocked after π. 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 α_k in π). The definition of an equivalent completion implies that t_k acquires and releases a lock on u during α_k in π_{cni}. Hence, t_k cannot acquire a lock on u again during β_k. Hence, u must be unlocked after π_{cni} as well. Thus, (t, e) is enabled after π_{cni}.

Now $\pi_{cni} \stackrel{t,e}{\rightarrow} \sigma'$ is an ACNI-execution, which satisfies DDL by assumption. So, the execution of (t, e) following π_{cni} satisfies DDL. The only interesting instruction here is a lock acquisition. If (t, e) acquires a lock on u after π_{cni}, u must be root. As explained above, the same condition must hold after π_{cni}. As a result, the execution of (t, e) after π must satisfy DDL as well.

Hence, (a) follows. As for (b), the ACNI-execution $\pi_{cni} \stackrel{t,e}{\rightarrow} \sigma'$ must have a completion by assumption. This gives us an equivalent completion for π'. (Lemma 28 above ensures that this is an equivalent completion.)

Case 2: Now, consider the case where $t = t_k$. Consider the schedule $\gamma = \alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_k$. (We don't consider β_k since t_k continues executing.) We now show that (t, e) is enabled after γ. Assume that (t, e) acquires a lock on u (following execution π). If u is the first lock acquired by t_k, then the reasoning is the same as in case 1. Otherwise, t_k must hold a lock on some predecessor x of u after π. This implies that none of the β_j, for $j < k$, can acquire a lock on x in the execution γ. This follows from the disjointness property described earlier (or, equivalently, from the definition of an equivalent completion). As a result, none of the β_j can acquire a lock on u either. Hence, u remains unlocked at the end of γ. Hence, (t, e) is enabled after γ.

Now $\gamma.(t, e)$ is an ACNI-execution, which satisfies DDL by assumption. We need to now show that $\pi.(t, e)$ satisfies DDL. It follows from Lemma 24 that we just need to check the cases where (t, e) acquires a lock. As explained above, none of the β_j in the execution of γ acquires a lock on u, Hence, they cannot insert or delete any edge $y \rightarrow u$. Thus, u has the same set of predecessors in the state after γ as in the state after π. Hence, the lock acquisition satisfies the DDL conditions after π iff it satisfies the conditions after γ.

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 π'. (The disjointness property above ensures that this is an equivalent completion.)

■

D. A Formal Definition of the Programming Model

In this section, we provide a formal definition of the programming language for concurrent modules introduced in Section 3.1.1.

D.1 Syntax

We represent the bodies of a procedures using control-flow graphs. We refer to the vertices of a control-flow graph as *program points*

Statement ($st = A(e)$)	Transition	Side Condition
`skip`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L\rangle]\rangle$	
`x = e(y1,...,yk)`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho[x \mapsto [\![e]\!](\rho(y_1), \ldots, \rho(y_k))], L\rangle]\rangle$	
`assume(b)`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L\rangle]\rangle$	$\rho(b) = tt$
`x = Y`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho[x \mapsto g(Y)], L\rangle]\rangle$	
`X = y`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g[X \mapsto \rho(y)], h, \varrho[t \mapsto \langle \kappa', \rho, L\rangle]\rangle$	
`acquire Y`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L \cup \{Y\}\rangle]\rangle$	$\forall \langle \kappa', \rho', L'\rangle \in range(\varrho) \bullet \mathtt{Y} \notin L$
`release Y`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L \setminus \{Y\}\rangle]\rangle$	$Y \in L$
`x.acquire()`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L \cup \{\rho(x)\}\rangle]\rangle$	$\forall \langle \kappa', \rho', L'\rangle \in range(\varrho) \bullet \rho(x) \notin L'$
`x.release()`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[t \mapsto \langle \kappa', \rho, L \setminus \{\rho(x)\}\rangle]\rangle$	$\rho(x) \in L$
`x = new R()`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h[a \mapsto \iota(R)], \varrho[\langle \kappa', \rho[x \mapsto a], L\rangle]\rangle$	$a \notin dom(h)$
`x = y.f`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h, \varrho[\langle \kappa', \rho[x \mapsto h(\rho(y))f], L\rangle]\rangle$	$\rho(y) \in dom(h)$
`x.f = y`	$\sigma \xrightarrow{\langle t,e\rangle} \langle g, h[\rho(x) \mapsto (h(\rho(x))[f \mapsto \rho(y)])], \varrho[\langle \kappa', \rho, L\rangle]\rangle$	$\rho(x) \in dom(h)$

Figure 5. Meaning of atomic statements. We use the following shorthands: $e = \langle \kappa, \kappa' \rangle$, $\sigma = \langle g, h, \varrho \rangle$, and $\langle \kappa, \rho, L \rangle = \varrho(t)$. We use $\iota(R)$ as a shorthand for an initialized record of type R. $\sigma \xRightarrow{\langle t,e\rangle} \sigma'$ c is a shorthand for $\sigma \xrightarrow{\langle t,e\rangle} \sigma'$ if c is true in σ and $\sigma \xrightarrow{\langle t,e\rangle} \bot$ otherwise.

$\kappa \in \mathcal{K}$. The edges of a control-flow graph are annotated with *primitive instructions*, shown in Figure 2.

To simplify the definition of the operational semantics, we assume that the syntactic domain of variable identifiers $x, X \in \mathcal{V} = \mathcal{V}_\mathcal{G} \uplus \mathcal{V}_\mathcal{L}$ is comprised of disjoint domains for global variables $X \in \mathcal{V}_\mathcal{G}$ and for local variables $x \in \mathcal{V}_\mathcal{L}$. We use uppercase letters for identifiers of the former and lowercase letters for identifiers for the latter.

Syntactic domains. We assume the unbounded syntactic domains of *program points* $\kappa \in \mathcal{K}$, *variable identifiers* $\mathcal{V} = \mathcal{V}_L \uplus \mathcal{V}_G$ comprised of *local variable identifiers* $x, y \in \mathcal{V}_L$ and *global variable identifiers* $X, Y \in \mathcal{V}_G$, *field identifiers* $f \in \mathcal{F}$, *record type identifiers* $\upsilon \in \mathcal{R}$, *procedure identifiers* $p \in \mathcal{P}$, and *module identifiers* $m \in \mathcal{M}$.

Concurrent modules. A *concurrent module* $cm = \langle m, tm, V, OP\rangle$ is comprised of a *module identifier* $id(cm) = m \in \mathcal{M}$; a set of *module type map* $\Gamma(cm) = tm \in \mathcal{TM}$ which associates *record type identifier* $\upsilon \in \mathcal{R}$ with a map from the fields of the record $f \in f(\upsilon) \subset_{fin} \mathcal{F}$ to their type. A type of a field may be either an atomic type (e.g., `int` or `bool`), or of a reference type to (an object of type) υ; a set $X, Y \in V(cm) = V \subset_{fin} \mathcal{V}_\mathcal{G}$ of *module-global variables*; and a set of *module procedures* $procs(cm) = OP$.

Module procedures. A *module procedure* $op = \langle p, V, G\rangle \in \mathcal{OP} = \mathcal{P} \times 2^\mathcal{V} \times CFG$ is comprised of a *procedure identifier* $id(op) = p \in FuncId$, a set $x, y \in V(p) = V \subset_{fin} \mathcal{V}_\mathcal{L}$ of *procedure-local variables* and a control-flow graph $cfg(op) = G$.

Control flow graphs. A *control flow graph* $G = \langle PP_G, E_G, A_G, \kappa^e_G, \kappa^x_G\rangle \in CFG = 2^\mathcal{K} \times 2^{\mathcal{K}\times\mathcal{K}} \times (\mathcal{K} \times \mathcal{K} \hookrightarrow Stmt) \times \mathcal{K} \times \mathcal{K}$ is a directed graph comprised of a set $PP_G \subset_{fin} \mathcal{K}$ of *program points*, a set $E_G \subset PP \times PP$ of *control-flow edges*, and a *statement map* $A_G : E_G \to Stmt$ which associates every control-flow edge $e \in E_G$ to one of the *primitive instructions* listed in Figure 2. G has two distinguished control points: an *entry site* $\kappa^e_G \in PP_G$, from which the procedure execution starts, and an *exit site* $\kappa^x_G \in PP_G$, in which the procedure execution ends.

D.2 Semantics

D.2.1 Memory States

Figure 3 defines the semantic domains of *memory states* and the meta-variables ranging over them. (See Section 3.1.2.)

Initial States. We simplify the operational semantics of a module by capturing the set of procedure invocations we would like to execute by the set of threads in the *initial* state. Thus, we assume that in the initial state the local state of every thread t records the procedure invoked by t (*i.e.*, the program counter of t is the entry node of the invoked procedure) as well as the parameters of the procedure invocation (as the initial values of the procedure local variables that represent the procedure parameters). All locks are free in the initial state (*i.e.*, no thread holds a lock). It is straightforward to extend the semantics to treat the concurrent module as a *reactive* system that responds to the arrival of procedure invocations, at any point during its execution, by adding a new thread with the appropriate initial state to its state.

D.2.2 Operational Semantics

In this section, we define the semantics domain and the operational semantics for an arbitrary module $cm = \langle m, V, OP\rangle \in \mathcal{CM}$.

We define the behavior of a concurrent module using a transition relation $\overset{tr}{\rightsquigarrow} \subseteq \mathcal{TR} = \Sigma \times \mathcal{T} \times (\mathcal{K} \times \mathcal{K}) \times \Sigma$ that interleaves the execution of different threads. (See Section 3.1.2.)

Small step operational semantics. Figure 5 defines a, rather standard, small step operational semantics pertaining to the primitive instructions in our language.

D.3 Operational Semantics

The behavior of a concurrent module is defined in Figure 5 by a transition relation $\overset{tr}{\rightsquigarrow} \subseteq \Sigma = \Sigma \times \mathcal{T} \times (\mathcal{K} \times \mathcal{K}) \times \Sigma \uplus \{\bot\}$ that interleaves the execution of different threads. We remind the reader that $\bot$ denotes a designated *error state*.