Static Analysis of Atomicity for Algorithms Using Non-Blocking Synchronization*

Liqiang Wang† and Scott D. Stoller ‡

Abstract

In concurrent programming, non-blocking synchronization is very efficient but difficult to design correctly. This paper presents a static analysis to show that code blocks using non-blocking synchronization are atomic, i.e., that every execution of the program is equivalent to one in which those code blocks execute without interruption by other threads. Our analysis determines commutativity of operations based primarily on how synchronization primitives (including locks, Load-Linked, Store-Conditional, and Compare-and-Swap) are used. A reduction theorem states that certain patterns of commutativity imply atomicity. Atomicity is itself an important correctness requirement for many concurrent programs. Furthermore, an atomic code block can be treated as a single transition during subsequent analysis of the program; this can greatly improve the efficiency of the subsequent analysis; for example, it can significantly reduce the time and space for model checking. We demonstrate the effectiveness of our approach on several concurrent non-blocking programs.

Categories and Subject Descriptors

[D.2.4] Software Engineering: Software/Program Verification

[D.1.3] Programming Techniques: Concurrent Programming

General Terms Verification, Algorithms, Design

Keywords Atomicity, Lock-free, Non-blocking, Synchronization, Static Analysis, Verification, Linearizability

1 Introduction

Many concurrent programs use blocking synchronization primitives, such as locks and condition variables. *Non-blocking synchronization primitives*, such as Compare-and-Swap, and Load-Linked / Store-Conditional, never block (*i.e.*, suspend execution of) a thread. Non-blocking (also called "lock-free") synchronization is becoming increasingly popular, because it offers several advantages, including better performance, immunity to deadlock, and tolerance to priority inversion and pre-emption [Mic04b, MS96].

An important use of non-blocking synchronization is in the implementation of non-blocking objects. A concurrent implementation of an object is *non-blocking* if it guarantees that some process can complete its operation on the object after a finite number of steps of the system, regardless of the activities and speeds of other processes [Her93]. Non-blocking synchronization is also used to implement blocking objects, such as spin locks.

*This work was supported in part by NSF under Grant CCR-0205376 and ONR under Grant N00014-02-1-0363. This paper significantly extends our previous work [WS05].

†Address: Department of Computer Science, University of Wyoming, Laramie, WY 82071-3315. Email: wang@cs.uwyo.edu.

‡Address: Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400. Email: stoller@cs.sunysb.edu.

Algorithms that use non-blocking synchronization are often subtle and difficult to design and verify. This paper presents a static analysis to show that code blocks using non-blocking synchronization are *atomic*. Informally, a code block is atomic if every execution is equivalent to one in which the code block is executed serially, *i.e.*, without interruption by other threads. Atomicity is well known in the context of transaction processing, where it is often called *serializability*.

Atomicity is an important correctness requirement for many concurrent programs. Furthermore, each atomic code block can be treated as a single transition during subsequent static or dynamic analysis of the program; this can greatly improve the efficiency of the subsequent analysis.

This paper presents a conservative intra-procedural static analysis to infer atomicity. We build on Flanagan *et al.*'s work on atomicity types [FQ03] and purity [FFQ05] in order to develop an analysis that is much more effective for programs that use non-blocking synchronization primitives. Our analysis first classifies all actions (*i.e.*, operations) in a program into different types based on their commutativity, which is determined based primarily on how locks and non-blocking synchronization primitives are used in the program. The analysis then combines those types to determine the types of larger code blocks, and then determines the atomicity of the program.

We formalize the analysis for a language SYNL that allows declaration of top-level procedures (as in an API) that implicitly get concurrently called by the environment. The language does not allow explicit procedure calls; internal procedures are inlined, and we do not handle recursion. Despite these limitations, SYNL is powerful enough for expressing many non-blocking algorithms. The analysis can be extended to be inter-procedural. It applies equally to non-blocking objects and blocking objects.

The analysis is incomplete (*i.e.*, sometimes fails to show atomicity), but is effective for common patterns of non-blocking synchronization, as demonstrated by the applications in Section 6. We applied it to four interesting non-trivial non-blocking programs: the running example in Sections 4 and 5, and three other programs described in Section 6. Our analysis can be applied directly to one of them. For the next two, we modify the algorithms slightly before applying our analysis and show that the modification preserves the behaviors of the algorithms. For the fourth one, we just omit some optional optimizations before applying our analysis.

We consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. We believe our analysis provides a useful method for manual verification of atomicity, as well as being suitable for automation. The experiments in Section 6 show that the atomicity analysis can significantly reduce the time and space of subsequent verification.

2 Related Work

This paper significantly extends our previous work [WS05] with a refined definition for pure loop, a formal semantics for the language SYNL, correctness proofs for core aspects of the analysis, and an analysis for DCAS-based double-ended queue.

Gao and Hesselink [GH04] used simulation relations to prove that a non-blocking (called lock-free in [GH04]) algorithm refines a higher-level (coarse-grained) specification. Using the PVS theorem prover, they proved correctness of algorithms similar to the ones in Figures 3 and 9. The proofs took a few man-months and are not easily re-usable for new algorithms.

Flanagan *et al.* developed type systems [FQ03] based on Lipton's reduction theorem [Lip75] to verify atomicity. Wang and Stoller [WS03, WS06b, WS06a] and Flanagan *et al.* [FF04] developed runtime algorithms to check atomicity. All of this work focuses on locks and is not effective for programs that use non-blocking synchronization.

Flanagan *et al.* extended their atomicity type system with a notion of purity [FFQ05]. A code block is pure if, whenever it evaluates without interruption by other threads and terminates normally, it does not change the program state. Non-blocking programs often contain code blocks that abort an attempted update to a shared variable if the variable was updated concurrently by other threads; these code blocks are often pure according to our definition of purity, which generalizes the definition in [FFQ05] by taking into account liveness of variables and use of unique references. The type system in [FFQ05] can show atomicity of simple non-blocking algorithms but not of any of the algorithms analyzed in this paper, because it does not accurately analyze usage of non-blocking synchronization primitives; for example, it has no analogue of the notions of "matching read" or "matching LL" in Section 5.2, and does not analyze exceptional variants (also defined in Section 5.2) of a procedure separately.

Atomicity used to optimize model checking can be regarded as a partial-order reduction [Pel98], *i.e.*, a method for exploiting commutativity to reduce the number of states explored by a verification algorithm. For non-blocking algorithms, traditional partial-order reductions are less effective than our analysis, because they do not distinguish left-movers and right-movers, and they focus on exploiting commutativity of operations with little regard for the context in which the operations are used, while our analysis considers in detail the context (surrounding synchronization and conditions) of each operation.

The model-checking (*i.e.*, state-space exploration) algorithm in [QRR04] dynamically identifies transactions, which correspond roughly to executions of atomic blocks. Their algorithm relies on a separate analysis to determine commutativity of actions. An inter-procedural extension of our analysis could be used for this. This would allow their algorithm to be applied effectively to non-blocking programs.

Linearizability [HW90] is a correctness condition for objects shared by concurrent processes. Informally, a concurrent object o is linearizable if and only if each concurrent operation history h for o is equivalent to some legal sequential history s, and s preserves the real-time partial order of operations in h. The equivalence is based on comparing the arguments and return values of procedure calls. Legality is defined in terms of a specification of the correct behaviors of the object. We focus on proving atomicity rather than linearizability, because atomicity does not require a specification of correct behaviors. Atomicity can help establish linearizability: first show that the concurrent implementation executed sequentially (*i.e.*, single-threaded) satisfies the sequential specification, and then apply our analysis to show that the procedures of the implementation are atomic.

3 Background

3.1 Non-Blocking Synchronization Primitives

Non-blocking synchronization primitives include *Load-Linked* (LL) and *Store-Conditional* (SC), supported by PowerPC, MIPS, and Alpha, and *Compare-and-Swap* (CAS), supported by IBM System 370, Intel IA-32 and IA-64, Sun SPARC, and the JVM in Sun JDK 1.5.

LL($addr$) returns the content of the given memory address. SC($addr, val$) checks whether any other thread has written to the address $addr$ (by executing a successful SC on it) after the most recent LL($addr$) by the current thread; if not, the new value val is written into $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and `false` is returned to indicate failure. Another primitive VL (validate) is often supported. VL($addr$) returns `true` iff no other thread has written to $addr$ after the most recent LL($addr$) by the current thread.

In a run of a program, the *matching* LL (if any) for a SC(v, val) or VL(v) action is the last LL(v) before that action in the same thread. If there is no matching LL for a SC action, the SC action fails. Similarly, the *matching* SC (if any) for a LL(v) is the next SC(v, val) after that action in the same thread.

CAS($addr, expval, newval$) compares the content of address $addr$ to the expected value $expval$; if the two values are equal, then the new value $newval$ is written to $addr$, and the operation returns `true` to indicate success; otherwise, the new value is not written, and the operation returns `false` to indicate failure.

3.2 A Language: SYNL

We formalize our analysis for a language SYNL (Synchronization Language). The syntax of SYNL is shown in Figure 1. There is no explicit procedure call, as discussed in Section 1.

A program consists of global variable declarations, thread-local variable declarations, and procedure definitions. A thread-local variable is accessible in only one thread, and its value persists between procedure calls of the same thread.

An execution of a SYNL program consists of an arbitrary number of invocations (by the environment) of its procedures with arbitrary type-correct arguments (for brevity, we leave the type system implicit), and with an arbitrary amount of concurrency. Therefore, SYNL does not need constructs to create threads.

Statements include assignments, `synchronized` (for lock synchronization), sequential composition, conditionals, `local` blocks, loops, `continue`, `return`, `break`, and `skip`. `synchronized` has the same semantics as in Java. A `local` statement introduces a scoped variable. In this paper, a local variable is introduced by a `local` statement, a `thread-local` declaration, or a procedure parameter declaration. The loop statement defines an unconditional loop: "`loop` s" is equivalent to "`while (true)` s". Every `while` loop can be re-written using `loop`, `if`, and `break`. All loops in SYNL are unconditional.

Expressions in SYNL include constant values, variables, field accesses, array accesses, non-blocking synchronizations, object allocations `new`, and calls to primitive operations (such as arithmetic operations). Variables may have primitive types and reference types. A local variable may contain a reference to a shared object. For example, a field access `x.fd` may access both a local variable `x` and a shared variable (*i.e.*, a field of a shared object). Primitive operations have no side effect.

As syntactic sugar, we allow non-blocking primitives to be used as statements when their return values are not needed; for example, $\mathrm{SC}(x, e)$ used as a statement is the syntactic sugar for: `local` $dummy = \mathrm{SC}(x, e)$ `in skip`.

$$\begin{array}{rcl}
\mathit{Program} & ::= & \texttt{global}\ \ \mathit{var}^*;\ \texttt{thread-local}\ \mathit{var}^*;\ \texttt{proc}\ \mathit{proc}^* \\
\mathit{Procedure} & ::= & \mathit{pn}(\mathit{var}^*)\ \mathit{stmt}^* \\
\mathit{Statement} & ::= & \mathit{loc} = e \mid \texttt{synchronized}(e)\ s \mid s; s \mid \texttt{if}\ e\ s\ s \mid \texttt{local}\ x = e\ \texttt{in}\ s \mid \texttt{loop}\ s \\
 & & \mid \texttt{continue} \mid \texttt{return} \mid \texttt{return}\ e \mid \texttt{break} \mid \texttt{skip} \\
\mathit{Expr} & ::= & \mathit{val} \mid \mathit{loc} \mid \texttt{CAS}(\mathit{loc}, e, e) \mid \texttt{LL}(\mathit{loc}) \mid \texttt{VL}(\mathit{loc}) \mid \texttt{SC}(\mathit{loc}, e) \mid \texttt{new}\ C \mid \mathit{prim}(e, ...) \\
\mathit{Location} & ::= & x \mid x.fd \mid x[e] \\
\mathit{proc} & \in & \mathit{Procedure} \\
s & \in & \mathit{Statement} \\
e & \in & \mathit{Expr} \\
\mathit{loc} & \in & \mathit{Location} \\
\mathit{pn} & \in & \mathit{ProcedureName} \\
\mathit{val} & \in & \mathit{Val} \\
x & \in & \mathit{Variable} \\
fd & \in & \mathit{Field} \\
\mathit{prim} & \in & \mathit{PrimitiveOperation} \\
C & \in & \mathit{Class}
\end{array}$$

Figure 1: Syntax of SYNL.

An *execution* is an initial state followed by a sequence of transitions. A program state is a tuple which consists of a global store G, a heap H, each thread's local store L and program statements (this indicates the next statement to execute; it takes the place of a program counter). Each transition corresponds to one step of evaluation of an expression or statement in a standard way. The formal definitions of states and transitions are in Appendix B. For each transition, we consider the action performed by it. These actions capture the relevant behavior of the transition for our analysis and are described in Section 3.3. Note that all constructs in SYNL are deterministic, so the intermediate states during an execution are uniquely determined by the initial state and the sequence of transitions, and we will sometimes talk about executions as if those states were present in them.

Code blocks in a program P are *atomic* if: for every reachable state s of P in which all threads are executing outside those code blocks, s is also reachable in an execution of P in which those code blocks are executed serially, *i.e.*, without interruption by other threads.

3.3 Commutativity and Atomicity Types

The *live scope* of a local variable consists of control points where the variable is in scope and live. We overload "live scope" by also saying that a state (of a multithreaded program) with some thread at such a control point is in the live scope of the variable. SYNL does not allow references to local variables, so the live scope of every local variable can easily be determined.

A local reference variable v in a program P is *unique* if, in every reachable state of P that is in the live scope of v, no other variables contain the same reference as v. Thus, during the live scope of v, the object pointed by v is unshared, although it may be shared outside the live scope of v. In this paper, "reference" is often short for "reference variable". Any static uniqueness analysis may be used to identify unique local references.

An *unshared object* is an object accessed by only one thread. An *unshared variable* is a local variable, a field of an unshared object, or a field of an object referenced by a unique local reference (this definition could be extended to include fields of objects reachable by a chain of unique references). Other variables are *shared variables*. Escape analysis can be used to determine when objects become shared; before that, their fields are unshared variables.

A *local action* is an access to an unshared variable. Conservatively, other variable accesses can be treated as *global actions*. Acquire and release on shared locks are also global actions. Thus, there are four kinds of global actions: read, write, acquire lock, and release lock. Let $R(v)$, $W(v)$, $acq(v)$, and $rel(v)$ denote these global actions, respectively, where v denotes the accessed variable or lock. LL and VL are global reads. SC and CAS are global writes to their first argument and, if their second or (for CAS) third argument are shared variables, also global reads of those variables. Every action is atomic, since a single action is executed in a single step of execution.

Following [Lip75], actions are classified according to their commutativity. An action is a *right-mover*/*left-mover* if, whenever it appears immediately before/after an action from a different thread, the two actions can be swapped without changing the resulting state. An action is a *both-mover* if it is a left-mover and a right-mover. An action not known to be a left-mover or right-mover is *atomic* (type), since a single action is executed in a single step of execution.

Theorem 3.1. *Local actions are both-movers.*

Proof. Accesses to local variables and fields of unshared objects are obviously both-movers. For accesses to a field f of an object o through a unique local reference v, they are all in the live scope of v. During the live scope of v, other threads cannot hold any reference to the object o. So they cannot access $o.f$ concurrently through v. Hence, accesses to $o.f$ through v are both-movers. □

Theorem 3.2. *Lock acquires are right-movers. Lock releases are left-movers.*

Proof. See [FQ03], from which this theorem is taken. Here is a proof sketch. For $acq(v)$, its immediate successor global action a from another thread can not be a successful $acq(v)$ or $rel(v)$, because $acq(v)$ would block, and $rel(v)$ would fail (in Java, it would throw an exception). Hence $acq(v)$ and a can be swapped without affecting the result, so lock acquire is a right-mover. For similar reasons, lock release is a left-mover. □

Theorem 3.3. *(1) For a global read $R(v)$, if no global write $W(v)$ from other threads can happen immediately before/after $R(v)$, $R(v)$ is a left/right mover. (2) For a global write $W(v)$, if no global read $R(v)$ or write $W(v)$ from other threads can happen immediately before/after $W(v)$, $W(v)$ is a left/right mover.*

Proof sketch. The main observations are that two reads commute, and accesses to different variables commute. □

We briefly review atomicity types, which were introduced by Flanagan and Qadeer [FQ03]. An atomicity type is associated with an action. The atomicity types are: *right-mover* (R), *left-mover* (L), *both-mover* (B), *atomic* (A), and *non-atomic* (N, called *compound* in [FQ03]). The first three mean that the actions have the specified commutativity. *Atomic* is used for single actions that are not left-movers or right-movers and for atomic (as defined in Section 3.2) sequences of actions. *Non-atomic* is used when none of the other atomicity types are known to apply. Atomicity types are partially ordered such that smaller ones give stronger guarantees. The ordering is: $B \sqsubset t \sqsubset A \sqsubset N$ for $t \in \{L, R\}$. The atomicity type of an expression or statement can be computed from the atomicity types of its parts (such as actions) using the following operations on atomicity types. The *join* (*i.e.*, least upper bound) operation based on this ordering is used to compute the atomicity of an `if` statement from the atomicity types of the `then` and `else` branches. The *iterative closure* t^* of an atomicity type t denotes the atomicity of a statement that repeatedly executed a sub-statement with atomicity type t. It is defined by: $B^* = B$, $R^* = R$, $L^* = L$, $A^* = N$, $N^* = N$. The *sequential composition* $a; b$ is defined by the following table (the rows are labeled by the first argument; the columns are labeled by the second argument):

;	B	R	L	A	N
B	B	R	L	A	N
R	R	R	A	A	N
L	L	N	L	N	N
A	A	N	A	N	N
N	N	N	N	N	N

4 Pure Loops

For a loop (recall that all loops in SYNL are unconditional, like `while (true)` s), if an iteration terminates exceptionally via a `break` or `return` statement, it is called an *exceptional iteration*; otherwise, it is called a *normal iteration*. For simplicity, we do not consider nested loops in this paper. We define pure loops based on the notion of pure statements introduced in [FFQ05][1]. Informally, a loop is *pure* if all normal iterations of the loop have no side effect (a formal definition appears in Section 4.1.1). Typically, a normal iteration uses some side-effect-free actions to check whether some conditions on the current state are satisfied. When these conditions are not satisfied, another iteration is needed. When these conditions are satisfied, an exceptional iteration occurs; it may have side effects and exit the loop. Therefore, following the idea proposed in [FFQ05], to determine the atomicity of a pure loop, we may ignore its normal iterations and focus on its exceptional iterations.

Note that pure is not the same as side-effect free, because a pure loop may have side effects in exceptional iterations.

A simple example of a pure loop appears in the implementation of the `Down` operation on a semaphore shown in Figure 2. Iterations that end at `return` are exceptional. Iterations that end at line 4 (*i.e.*, when `tmp > 0` is false) or line 5 (*i.e.*, when SC returns `false`) are normal and have no residual side effects.

[1]In our framework, unlike [FFQ05], purity is a property (of loops) that has no effect on the operational semantics.

```
proc Down(sem) {
  loop
    local tmp = LL(sem) in            local tmp = LL(sem) in
      if (tmp > 0)                      TRUE(tmp > 0)
        if (SC(sem, tmp-1))             TRUE(SC(sem, tmp-1))
          return;                       return;
}
```

Figure 2: `Down` operation on a semaphore. Its exceptional variant introduced in Section 5.2.1 is shown on the right.

4.1 Formal Definition of Pure Loops

4.1.1 Pure Actions and Pure Loops

Some local reference is not unique according to the definition in Section 3.3, but all its effects on other threads are limited to normal iterations of loops, which can be ignored under some condition when analyzing atomicity, as shown below in Theorem 4.1. To capture this, we define: a local reference variable v is *quasi-unique* in a program P if v is unique (as defined in Section 3.3) when all *loop-local* variables (*i.e.*, their entire scopes are in the loop body) in normal iterations of loops in P are ignored (in other words, some loop-local variables in normal iterations may contain the same reference as v while v is in scope and live). In this paper, quasi-unique local references are identified manually. This could be done automatically using an extended static uniqueness analysis.

Informally, an action in a normal iteration of a loop is *pure* if any update performed by the action is not visible to other threads or to the current thread after the current normal iteration; in other words, there is no data flow from the action to outside of the normal iteration in which it occurs. Formally, a pure action should satisfy the following two conditions:

(1) If it performs an update, the target location *loc* must satisfy the following conditions: (*1.i*) *loc* is a local variable, a field of an unshared object, or a field of an object o accessed through a quasi-unique local reference; and (*1.ii*) for all paths in the control flow graph from the beginning of the loop body to exit points of the loop (*i.e.*, `break` and `return` statements), if there is any access to *loc*, the first one must be write (and if *loc* is a field of a shared object, the write is performed by dereferencing a unique local reference); and (*1.iii*) if *loc* is not accessed on some path described in (*1.ii*), then *loc* is loop-local.

(2) (*2.i*) If it is a LL action, each matching SC(*loc*,-) is in the same iteration of the loop; if it is a SC action, each matching LL(*loc*) is also in the same iteration. (*2.ii*) It is not a successful SC.

Condition (1) as a whole ensures that the updates in normal iterations have no side-effects outside. Condition (*1.i*) ensures that the updates in normal iterations effectively have no side-effects on other threads. Suppose a field f of a shared object o is updated in a normal iteration through a quasi-unique local reference, other threads concurrently access $o.f$ (and hence see the update) only in normal iterations, so those accesses and their effects also disappear when normal iterations are deleted, as in Theorem 4.1. Conditions (*1.ii*) and (*1.iii*) ensure that updates in normal iterations are not visible to subsequent operations of the same thread. Condition (*2.i*) is needed because LL implicitly performs an update that can affect subsequent SC operations by the same thread. Condition (*2.ii*) is needed because a successful SC has side-effects on SC operations in other threads.

Formally, a loop is *pure* if, for each normal iteration of the loop, every action that can occur in it is pure. To check whether a loop is pure, we construct a control flow graph (CFG), analyze it to identify actions that can occur in normal iterations of the loop, and then check whether those actions are pure according to the

```
global Q;
thread-local prv;
proc alg(input)
1   loop
2     local m = LL(Q) in
3       copy(prv.data,m.data);
4       if (!VL(Q)) continue;
5       computation(prv.data,input);
6       if (SC(Q,prv))
7         prv = m;
8         break;
```

```
local m = LL(Q) in
  copy(prv.data,m.data);
  TRUE(VL(Q));
  computation(prv.data,input);
  TRUE(SC(Q,prv));
  prv = m;
  break;
```

Figure 3: Herlihy's non-blocking algorithm for small objects. Its exceptional variant, defined in Section 5.2.1, is shown on the right.

above definition. There is a special case for SC and CAS. When a SC is used as the test condition of an `if` statement (*e.g.*, the SC in Figure 3), if only the false branch of the `if` statement can be executed under normal iterations, the SC is treated as a read (not an update). CAS is handled similarly.

4.1.2 An Example of Pure Loop

An example appears in Figure 3, which shows Herlihy's algorithm for non-blocking concurrent implementation of small objects [Her93]. Suppose a small object (*i.e.*, small enough to be copied efficiently) is shared by a set of threads. The main steps on each thread in the algorithm are: (1) read the shared object reference Q using LL; (2) copy the data from the shared object into a private (*i.e.*, currently unshared) working copy of the object, *i.e.*, the object referred by *prv*; (3) perform the requested computation on the private object; (4) switch the reference values in Q and *prv* between the shared object and the private object using SC and an assignment statement. Note that the formerly shared object becomes a private copy, and the formerly private object becomes the current shared copy.

Before a thread t_1 switches the reference of the shared copy o with the private copy of t_1, another thread t_2 may read the reference to o using `LL(Q)`. Even though o becomes the private copy of t_1, t_2 may still hold the reference to o, though the SC of t_2 will fail later, causing t_2 to loop and read the current reference from `Q`. Thus, t_1 may write to o while t_2 copies data from o. If t_2 tried to perform a computation on a copy of the data that reflects only part of some update, it might suffer a fatal error, such as divide by zero. Line 4 prevents this: if *o.data* (accessed as `m.data`) is modified by another thread during the copy in line 3, the VL will fail.

Initially, *prv* is a unique local reference. After the reference values in Q and *prv* are switched, some other threads executing the loop may still have references to the formerly shared object as described in the previous paragraph. Note that *prv* can be aliased only by m and Q variables at those threads during normal iterations. Therefore, when normal iterations are ignored, *prv* is a unique local reference except in the states between a successful SC and "$prv = m$" because the thread temporarily has two references to the object. Since *prv* and m are both local variables, "$prv = m$" contains only local actions. It is well known that the set of reachable states is not affected when a sequence of local actions executes atomically together with a non-local action that immediately precedes or immediately follows the sequence, if the intermediate states between those actions are ignored. In this case, ignoring those intermediate states is justified because these transitions are in a loop body, and our goal (shown in Theorems 4.1 and 4.2) is to preserve reachability of states where all threads are executing outside of loops (because the theorems require that all loops are pure).

Therefore, according to the definition in Section 4.1.1, all actions in normal iterations are pure, so the loop is pure.

4.1.3 Normal Iterations of Pure Loops Can Be Deleted

Theorem 4.1 shows that all normal iterations of all pure loops can be deleted from an execution without affecting the result of the execution. Informally, deleting a transition from an execution means removing it and adjusting the subsequent states. Details are in Appendix C. Theorem 4.1 is the basis for proving in Section 5 that normal iterations of pure loops can be ignored when analyzing atomicity.

Theorem 4.1. *Suppose all loops in a program P are pure. Let σ be an execution of P. Let σ' be an execution obtained from σ by deleting all transitions in all normal iterations of all pure loops in P. Then σ' is also an execution of P, and σ and σ' contain the same states in which all threads are executing outside pure loops.*

Proof sketch. A detailed proof appears in Appendix C. Here is a proof sketch.

Recall that loops in SYNL are equivalent to `while (true)` s. When the body of a loop terminates normally, the thread begins another iteration of the same loop body.

According to the definition of pure loop, normal iterations can update only unshared variables and fields of shared objects accessed by dereferencing quasi-unique local references. The values written are dead by the end of the loop body, except that an update to a field of a shared object accessed by dereferencing a quasi-unique local reference can be visible in normal iterations of pure loops in other threads. Because all normal iterations of pure loops are removed simultaneously, all of these side-effects on other threads are also eliminated. The syntax of SYNL ensures that lock acquire and release actions occur in matching pairs in an execution of a loop body, so deleting them does not affect the resulting state of the lock; also, note that other threads cannot perform any operations on a lock while it is held by this thread. Because the definition of pure loop implies that no successful SC occurs in normal iterations, and the matching pairs of LL and SC (if they exist) must occur in the same iteration of a loop, so deleting them does not affect subsequent LLs and SCs in this thread or LLs and SCs in other threads. CAS just reads and writes variables, and the same reasoning as for other reads and writes applies. □

Theorem 4.2. *Suppose all loops in a program P are pure. After deleting all transitions in all normal iterations of all loops, accesses to fields of shared objects by dereferencing quasi-unique local references are both-movers.*

Proof. According to the definition of quasi-unique local references, these references become unique after all normal iterations of all loops are deleted. As shown in Theorem 3.1, accesses through unique local references are both-movers. □

5 Checking Atomicity

The main issue in applying Theorem 3.3 is determining whether a global action can happen immediately before or after another global action. Our analysis determines this based on how synchronization primitives are used.

5.1 Lock Synchronization

Lock synchronization is well studied. We sketch a simple treatment of lock synchronization, to illustrate how analysis of locks fits into our overall analysis algorithm.

Theorem 5.1. *If expressions e_1 and e_2 appear in the bodies of different* `synchronized` *statements that synchronize on the same lock, then e_1 cannot be executed immediately before or after e_2.*

Proof sketch. Since e_1 and e_2 are protected by the same lock, at least one acquire and release of the lock must occur between e_1 and e_2 in any execution. □

Alias analysis may be used to determine whether two `synchronized` statements synchronize on the same lock.

5.2 Non-Blocking Synchronization

5.2.1 Exceptional Variants

Based on Theorem 4.1, for each pure loop, it suffices to analyze atomicity of each exceptional iteration of the loop. For each `break` or `return` statement in a loop, the backward slice of the loop body starting at that `break` or `return` and ending at the loop's entry point is called an *exceptional slice* of the loop.

The atomicity of a procedure can be determined by analyzing atomicity of its exceptional variants. Each *exceptional variant* is a specialized version of the procedure, and corresponds to a selection of exceptional slices of its pure loops, with each pure loop replaced by its selected exceptional slice. If the selected exceptional slice includes only the true branch of an "`if` e S_1 S_2" statement, then we replace the `if` statement with "`TRUE(`e`)`; S_1" in the corresponding exceptional variants of the procedure; if the slice includes only the false branch, we replace the `if` statement with "`TRUE(`$!e$`)`$; S_2$". A SC expression in `TRUE(SC(`v, val`))` must be successful, and we call it a *successful* SC expression. Non-pure loops appear unchanged in the exceptional variants. For example, Figures 2 and 3 show the exceptional variants for the semaphore `Down` procedure and Herlihy's non-blocking algorithm, respectively.

Theorem 5.2. *If all exceptional variants of a procedure p are atomic, then p is atomic.*

Proof Sketch. Let P denote the original program that contains p. Let σ be an execution of P. Let φ be a state in σ in which all threads are executing outside p.

According to Theorem 4.1, an execution σ' of P can be obtained from σ by deleting all transitions in normal iterations of pure loops in procedure p, and φ is reachable in σ'. By the definition of exceptional variant, there must be exceptional variants of p which can produce the same exceptional iterations of pure loops of p as in σ'. Let P' denote the program obtained by replacing procedure p with such exceptional variants. Thus, σ' is also an execution of P'.

By hypothesis, all exceptional variants of p are atomic. Based on σ', by the definition of atomicity, there exists an execution σ'' of P' in which the exceptional variant of p are executed atomically and in which φ is reachable.

By the definition of exceptional variants of a procedure, every execution of an exceptional variant of p is also an execution of p. Therefore, σ'' is also an execution of P, and all executions of p in σ'' are executed atomically, and φ is reachable in σ''. Thus, by the definition of atomicity, p is atomic. $\square$

5.2.2 Atomicity Analysis of Non-Blocking Synchronization Primitives LL/SC/VL

There is a unique matching LL action for each successful SC action in an execution. In program code, there might be multiple LL expressions or statements that can produce the matching LL action for an occurrence of SC. We call these the *matching* LL expressions of the SC expression. For example, if there is an `if` statement before a SC, and both branches of the `if` statement contain LL, both of the LL expressions can possibly match the SC.

For a SC(v, val) in a program, to find its matching LL expressions, we do a backward depth-first search on the control flow graph starting from the SC, and not going past edges labeled with LL(v). All of the visited occurrences of LL(v) match the SC. For a VL(v), its matching LLs can be found in the same way.

We implicitly assume hereafter that each SC expression has a unique matching LL expression. This assumption is not essential, but it simplifies the analysis and is satisfied by the non-blocking algorithms we have seen. We also implicitly assume that a variable updated by a SC is updated only by SC, not by other actions (such as regular assignment or CAS).

Theorem 5.3. *A successful SC or VL is a left-mover, and the matching LL is a right-mover.*

Proof. By the semantics of LL, SC and VL, for a successful SC(v, val) or VL(v) action and its matching LL(v), any other SC action (successful or failed) on v executed by another thread cannot be executed between them. Therefore, the successful SC or VL is a left-mover, and the matching LL is a right-mover. □

Theorem 5.4. *Let SC and LL denote a successful SC(v, val) expression and its matching LL(v) executed by a thread t, respectively. Let SC' and LL' denote a successful SC(v, val') expression and its matching LL(v) executed by another thread t', respectively. SC', LL', and all transitions of t' between them cannot be executed between SC and LL.*

Proof sketch. According to Theorem 5.3, SC' cannot happen between LL and SC. Thus, there are two cases: SC' happens before LL, or SC' happens after SC. For the first case, the theorem obviously holds. For the second case, if LL' happens between LL and SC, SC must also happen between LL' and SC' (because SC succeeds), which is impossible since SC' also needs to succeed; if LL' happens after SC, the theorem obviously holds. □

5.2.3 Atomicity Analysis of Non-Blocking Synchronization Primitive CAS

CAS is often used in a similar way as LL/SC. CAS takes an address, an expected value, and a new value as arguments. There is often an assignment before CAS to save the old value into a temporary variable that is used as the expected value. For a CAS, its *matching read*, if any, is the action which reads the old value and saves it as the expected value. Note that a CAS can succeed even without a matching read; a SC cannot succeed without a matching LL. We use a backward search on the control flow graph to find the matching reads for a CAS expression. We implicitly assume hereafter that there is a unique matching read for each CAS.

CAS-based programs may suffer from the ABA problem: if a thread reads a value A of a shared variable v, computes a new value A', and then executes CAS(v, A, A'), the CAS may succeed when it should not, if the shared variable's value was changed from A to B and then back to A by CASs of other threads. A common solution is to associate a modification counter with each variable accessed by CAS [Mic04b]. The counter is read together with the data value, and each CAS checks whether the counter still has the previously read value. A successful CAS increments the counter. With this mechanism, variants of Theorem 5.3 and 5.4 hold for CAS: just replace "matching LL" with "matching read", and replace "SC" with "CAS".

5.3 Non-Blocking Synchronization and Invariants

In many algorithms, an invariant holds throughout execution of a code block. If the invariants for two code blocks contradict, executions of the two code blocks by different threads cannot be interleaved.

A predicate $p(lvar)$ is called a *local invariant* of a code block "`local` $lvar = e$ `in` *stmt*" (which is called a *local block* on $lvar$), if it satisfies the following two conditions: (i) $lvar$ is not updated in *stmt*, and (ii) $p(lvar)$ holds throughout execution of *stmt*.

Condition (i) is easy to check, because there is no aliasing of local variables in SYNL. When condition (i) holds, a local invariant for a block can easily be obtained from the `TRUE` statements in *stmt* that depend only on $lvar$. For example, in the exceptional variant of procedure `Down` shown in Figure 2, a local condition for the code block in lines 3-6 is $tmp > 0$. If condition (i) does not hold, or no appropriate `TRUE` statements appear in the local block, its local invariant is `true`.

A local block of the form "`local` $lvar$ = `LL`($svar$) `in` {$stmt$;`TRUE(SC(`$svar, val$`));`}" is called a *LL-SC block* on $svar$.

Theorem 5.5. *Suppose a shared variable svar is updated only by SC expressions in LL-SC blocks, and every LL-SC block on svar in the program has the same local invariant $p(lvar)$. Suppose a local block S "`local` $lvar' = svar$ `in` $stmt'$" has a local invariant $!p(lvar')$.*

(a) No successful SC(svar) in the LL-SC blocks on svar can occur inside S.

(b) No transition in local block S can be executed inside any LL-SC block on svar, and no transition in any LL-SC block on svar can be executed inside local block S.

Proof of (a). We prove (a) by contradiction. Let "`local` $lvar$ = `LL`($svar$) `in` {$stmt$;`TRUE(SC(`$svar, val$`));`}" be a LL-SC block on $svar$. Suppose a successful SC($svar$) in this LL-SC block occurs inside S; this implies that S and the LL-SC block are being executed by different threads. Without loss of generality, we consider the first such SC($svar$). According to the assumption for the local block S, $!p(lvar')$ holds during $stmt'$. Because $svar$ is updated only by SC actions from LL-SC blocks, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until SC($svar$) is executed. This implies that $!p(svar)$ holds when SC($svar$) is executed. The LL-SC block has local invariant $p(lvar)$, and $lvar == svar$ holds until the SC, because $lvar$ is not updated in the LL-SC block, and $svar$ is not updated before the first successful SC on it, so $p(svar)$ holds when SC($svar$) is executed. This contradicts the previous conclusion.

Proof of (b). According to the proof of (a), no successful SC($svar$) can happen inside S. Consider an execution of a LL-SC block on $svar$. There are two cases:

Case 1: the successful SC happens before S. Thus, the whole LL-SC block happens before S. Obviously, the theorem holds in this case.

Case 2: the successful SC happens after S. If the matching LL also happens after S, the whole LL-SC block happens after S. Hence the theorem holds. Suppose the matching LL happens inside S or before S. Similar to the proof of (a), because $svar$ is updated only by SC actions from LL-SC blocks, and no successful SCs happen inside S or between the matching LL and the SC, $lvar' == svar$ and hence $!p(svar)$ holds from the start of $stmt'$ until the SC($svar$) happens. Thus, $!p(svar)$ holds when SC($svar$) happens. By the same reasoning as in the proof of (a), $p(svar)$ holds when SC($svar$) happens. This contradicts the previous conclusion. Therefore, when SC happens after S, the matching LL cannot happen inside S or before S. $\square$

The definition of LL-SC block and the above theorem can be generalized, so that the LL does not need to occur at the beginning of a local block, and the SC does not need to occur at the end of a local block. A similar theorem holds for CAS.

5.4 Atomicity Inference

To analyze atomicity of each procedure in a program, we identify pure loops, then check atomicity of each procedure's exceptional variants, by computing atomicity types for all expressions and statements, as follows:

- Step 1: Identify all local actions and lock actions. According to Theorem 3.1, all local actions have atomicity type B. According to Theorem 3.2, all lock acquires and releases have atomicity type R and L, respectively.

- Step 2: According to Theorem 5.3, if all updates on a variable v are done through SC, all successful SC(v, val) and VL(v) have atomicity type L, and their matching LL(v) have atomicity type R. A successful VL(v) between a successful SC(v, val) and the matching LL(v) has atomicity type B. The analogous theorem for CAS is used for successful CAS and the matching reads.

- Step 3: Infer local invariants for local blocks, as described in Section 5.3.

- Step 4: Using Theorems 5.1, 5.4 and 5.5, for each read, check whether there is a write on the same variable that can happen immediately before/after it; for each write, check whether there is a read or write on the same variable can happen immediately before/after it. For access to variables on the heap, the analysis does a case split on whether two field accesses refer to the same location; both cases are considered, unless alias analysis shows one is impossible. Our current alias analysis just checks whether the references have the same type and whether the accessed fields have the same name; if not,

```
local m = LL(Q) in                a1:R local m = LL(Q) in
  copy(prv.data,m.data);          a2:B   copy(prv.data,m.data);
  if (!VL(Q)) continue;           a3:B   TRUE(VL(Q));
  computation(prv.data,input);    a4:B   computation(prv.data,input);
  if (SC(Q,prv))                  a5:L   TRUE(SC(Q,prv))
    prv = m;                      a6:B   prv = m;
    break;                        a7:B   break;
```

Figure 4: Atomicity types for actions of Herlihy's non-blocking algorithm.

the two field accesses must be to different locations. Assign atomicity types to the reads and writes based on Theorem 3.3, if they were not given atomicity types in previous steps.

- Step 5: For actions not given an atomicity type in previous steps, conservatively assign them atomicity type A.
- Step 6: Propagate atomicity types from the actions up through the abstract syntax trees of the procedures using the atomicity calculus sketched in Section 3.3 and detailed in [FQ03]. The atomicity type of a compound program construct is computed from the atomicity types of its parts using join, sequential composition, and iterative closure, as appropriate.

At last, for each procedure *p* in the original program, if every exceptional variant of *p* has atomicity type A, then by Theorem 5.2, *p* has atomicity type A.

As an example, we compute the atomicity of Herlihy's non-blocking algorithm for small objects as shown in Figure 4. Recall that the procedure has one exceptional variant, given in Section 5.2.1. In step 1, local actions in line a6 and line a7 are identified and assigned atomicity type B. Since *prv* is a quasi-unique local reference, the computation action in line a4 and the copy action in line a2 (the read in line a2 will consider later) are both-movers according to Theorem 4.2, hence, they are assigned atomicity type B. In step 2, successful SC in line a5 has atomicity type L, the matching LL in line a1 has atomicity type R, and successful VL in line a3 has atomicity type B. Step 3 is skipped in this algorithm. In step 4, we know any write to `m.data` (which must through successful SC) in other threads cannot happen immediately before or after the read to `m.data` in line a2, since a successful SC is followed. Hence, both actions in line a2 have atomicity type B. Now we can propagate atomicity types from the actions, and conclude the only exceptional variant is atomic. Therefore, the procedure is atomic.

6 Applications

This paper demonstrates the applicability of our analysis to four non-trivial non-blocking algorithms from the literature. One is the running example, *i.e.*, Herlihy's non-blocking algorithm; the other three are presented in this section. Although in two cases we must modify the algorithms slightly, and in one case we omit some optimizations before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatic) analysis that can show atomicity of the modified or original versions.

6.1 Michael and Scott's Non-Blocking FIFO Queue Using LL/SC/VL

6.1.1 NFQ and NFQ′

Figure 5 contains code for a non-blocking FIFO queue (NFQ) that uses LL/SC/VL [Mic04a]. It is similar to the well-known CAS-based algorithm in [MS96]. It uses a singly-linked list whose head and tail are pointed

```
proc Enq(value)
  local node = new Node();
    node.value = value;
    node.next = null;
    loop
      local t = LL(Tail) in
        local next = LL(t.Next) in
          if (!VL(Tail)) continue;
          if (next != null)
            SC(Tail,next);
            continue;
          if (SC(t.Next,node))
            // optional
            [SC(Tail,node);]
            return;
```

```
proc Deq()
  loop
    local h = LL(Head) in
      local next = h.Next in
        if (!VL(Head)) continue;
        if (next == null)
          return EMPTY;
        if (h == LL(Tail))
          SC(Tail,next);
          continue;
        local value = next.Value in
          if (SC(Head,next))
            return value;
```

Figure 5: Michael and Scott's Non-Blocking FIFO Queue (NFQ). `Head` and `Tail` are global variables.

to by global variables `Head` and `Tail`. Enqueue consists of three main steps: create a node, add it to the end of the list, and update `Tail`. A blocking implementation would use a lock around the second and third steps to achieve atomicity. In the non-blocking algorithm, if a thread gets delayed (or killed) after the second step, other threads may update `Tail` on its behalf; in that case, if the delayed thread later tries to update `Tail`, its SC will harmlessly fail. To avoid blocking, the dequeue operation also updates `Tail`. Dequeue is also non-blocking. A dummy node is used as the head of the queue to avoid degenerate cases. The code for `Deq` in [Mic04a, MS96] stores the value of `LL(Tail)` in a local variable; the code in Figure 5 does not. This does not affect the correctness or performance of the algorithm but makes it easier to analyze.

We would like to show that NFQ is linearizable, using the two-step approach described in Section 2: one step is to show that the concurrent implementation executed sequentially satisfies the sequential specification; the other step is to apply our analysis to show that the procedures of the implementation are atomic.

An obstacle to apply our atomicity analysis to NFQ is that the loops in `Enq` and `Deq` are not pure, because of the updates to `Tail` in normal iterations. Therefore, we modify the program to make the loops pure before applying our analysis algorithm; specifically, we consider the modified program NFQ′ in Figure 6, and we prove in Appendix A that the modification preserves linearizability. In NFQ′, all updates to `Tail` are performed in a separate procedure `UpdateTail`. `UpdateTail` may be invoked (by the environment) at any time, so NFQ′ is effectively more non-deterministic than NFQ.

6.1.2 Atomicity of NFQ′

All exceptional variants for the procedures of NFQ′ are listed in Figure 7. Each line of code is labeled on the left with a line number and the atomicity type of the code on that line. A line may contain multiple actions; we refer to the sequential composition of their atomicity types as the atomicity type of the line. Next we describe how the atomicity analysis algorithm in Section 5.4 works on these procedures.

In step 1, a1, a2, a3, a7, a9, b4, b6, c4, c5, d4 and d8 are classified as both-movers, because they access local variables.

In step 2, a4, a5, b1, c1 are d1 are classified as right-movers, because they are matching LLs for successful SCs or VLs; a6 (which is reclassified as a both-mover in step 4), a8, b5, c3, and d7 are classified as left-movers because they are successful SCs or VLs; b3 and d3 are classified as both-movers because they are between matching LLs and successful SCs.

In step 3, the local invariant for a5-a9 and c2-c5 is $next == null$. The local invariant for b2-b6 and d2-d8 is $next \,!= null$.

```
proc AddNode(value)
  local node = new Node() in
    node.Value = value;
    node.Next = null;
    loop
      local t = LL(Tail) in
        local next = LL(t.Next) in
          if !VL(Tail)
            continue;
          if (next != null)
            continue;
          if SC(t.Next,node)
            return;
```

```
proc UpdateTail()
  loop
    local t = LL(Tail) in
      local next = t.Next in
        if !VL(Tail)
          continue;
        if (next != NULL)
          SC(Tail,next);
          return;
```

```
proc Deq'()
  loop
    local h = LL(Head) in
      local next = h.Next in
        if (!VL(Head))
          continue;
        if (next == null)
          return EMPTY;
        if (h == LL(Tail))
          continue;
        local value = next.Value in
          if (SC(Head,next))
            return value;
```

Figure 6: NFQ′, a modified version of NFQ.

Now consider step 4. Let $\mathtt{t}_a$ and $\mathtt{t}_u$ denote the local variable `t` in `AddNode` and `UpdateTail`, respectively. If $\mathtt{t}_a$`.Next` of the LL-SC block in `AddNode` is aliased with $\mathtt{t}_u$`.Next` of the local block in `UpdateTail`, then according to Theorem 5.5, the update on `Tail` (*i.e.*, b5) cannot happen between a6 and a7, so a6 is a both-mover. a8 cannot happen between b2 and b3, so b2 is a right-mover. Suppose $\mathtt{t}_a$`.Next` is not aliased with $\mathtt{t}_u$`.Next`; this implies $\mathtt{t}_a$ is not aliased with $\mathtt{t}_u$, *i.e.*, $\mathtt{t}_a \neq \mathtt{t}_u$, so even if a8 happens between b2 and b3, b2 is a right-mover by Theorem 3.3. $\mathtt{t}_a \neq \mathtt{t}_u$ implies that the value of `Tail` read in line of a4 in `AddNode` is not equal to the value of `Tail` read in line of b1 in `UpdateTail`. Thus, even if b5 happens between a6 and a7, a6 is still a both mover by Theorem 3.3. For d2, if `h.Next` is aliased with `t.Next` of `AddNode`, a8 cannot happen between d2 and d3 according to Theorem 5.5, hence d2 is a right-mover by Theorem 3.3; if `h.Next` is not aliased with `t.Next`, d2 is again a right-mover. Also in step 4, d6 is inferred to be a both-mover, because there is no write to the `Value` field of any shared object; the only write a2 to the `Value` field is on an object that has not escaped.

In step 5, the unclassified c2 and d5 are given atomicity type A. Step 6 infers that each procedure in Figure 7 has atomicity type A. Step 7 infers that all procedures in NFQ′ are atomic.

6.1.3 Linearizability of NFQ′ and NFQ

We showed in Section 6.1.2 that the procedures in NFQ′ are atomic, and we showed in Appendix A that NFQ′ can simulate all behaviors of NFQ. To conclude that NFQ′, and hence NFQ, are linearizable with respect to a sequential specification of FIFO queues, we need to show that NFQ′ executed sequentially satisfies that specification. One approach is to use a powerful verification tool such as TVLA [YS03], which is a model checker based on static analysis. With our approach, TVLA only needs to consider sequential executions of

```
proc AddNode(value)
a1:B   local node = new Node() in
a2:B     node.Value = value;
a3:B     node.Next = null;
a4:R     local t = LL(Tail) in
a5:R       local next = LL(t.Next) in
a6:B         TRUE(VL(Tail));
a7:B         TRUE(next == null);
a8:L         TRUE(SC(t.Next,node));
a9:B         return;
```

```
proc UpdateTail()
b1:R   local t = LL(Tail) in
b2:R     local next = t.Next in
b3:B       TRUE(VL(Tail));
b4:B       TRUE(next != NULL);
b5:L       TRUE(SC(Tail,next));
b6:B       return;
```

```
proc Deq'1()
c1:R  local h = LL(Head) in
c2:A    local next = h.Next in
c3:L      TRUE(VL(Head));
c4:B      TRUE(next == null);
c5:B      return EMPTY;
```

```
proc Deq'2()
d1:R  local h = LL(Head) in
d2:R    local next = h.Next in
d3:B      TRUE(VL(Head));
d4:B      TRUE(next != null);
d5:A      TRUE(h != LL(Tail));
d6:B      local value = next.Value in
d7:L        TRUE(SC(Head,next));
d8:B        return value;
```

Figure 7: Exceptional variants for procedures of NFQ′.

program	without atomic		with atomic	
	states	time	states	time
unbounded `AddNode` threads	4500	> 19h	13	3.0s
unbounded `Deq'` threads	1285	88 m	10	1.7s
incorrect `AddNode`	13	5 s	13	3.0s

Table 1: Experimental results for verification of NFQ′ with TVLA.

NFQ′, so the verification is much faster and use much less memory than the verification in [YS03], where TVLA was used to show directly that NFQ satisfies some complicated temporal logic formulas.

To evaluate the speedup that our atomicity analysis can provide for subsequent verification, following [YS03, Table 2], we used TVLA to verify several correctness properties of NFQ′. We analyzed the correct program with two different environments: in the first one, the number of threads that concurrently call `AddNode` is unbounded (but there is only one thread that performs dequeues, and there is only one `UpdateTail` thread, since it contains a non-terminating loop); in the second one, the number of threads that concurrently perform dequeues is unbounded (but there is only one thread that performs `AddNode`, and one that calls `UpdateTail`). We also checked the properties for an incorrect version of NFQ′; specifically, we deleted the statement "`if (next != null) continue`" in the `AddNode` procedure; TVLA catches this error. We performed all experiments twice: once with each procedure body declared as atomic, as inferred by our analysis algorithm, and once without those declarations. The atomicity declarations had little effect on the time needed for TVLA to find an error in the incorrect program, but it reduced the time and space needed to verify the correct versions by a factor of 100 or more. The experimental results appear in Table 1.

6.2 Gao and Hesselink's Non-Blocking Algorithm for Large Objects

For large objects, copying is the major performance bottleneck in Herlihy's algorithm. Gao and Hesselink [GH04] proposed an algorithm that avoids copying the whole object. The fields of each object are divided into `W` disjoint groups such that each operation changes only fields in one group. When copying data

```
proc alg1(SharedObj, g)
  loop
    local m = LL(SharedObj) in
      copy(prvObj.data[1], m.data[1]);
      if (!VL(SharedObj)) continue;
      ... // repeat the previous two lines W times,
          // incrementing the index each time,
          // to copy each group of fields
      compute(prvObj, g);
      if (SC(SharedObj, prvObj))
        prvObj = m;
        return;
```

```
proc alg2(SharedObj, g)
  loop
    local m = LL(SharedObj) in
      if (prvObj.data[1] != m.data[1])
        copy(prvObj.data[1], m.data[1]);
        if (!VL(SharedObj)) continue;
      ... // repeat the previous three lines W times,
          // incrementing the index each time,
          // to copy each group of fields.
      compute(prvObj, g);
      if (SC(SharedObj, prvObj))
        prvObj = m;
        return;
```

Figure 8: Gao and Hesselink's non-blocking algorithm for large objects: Algorithms 1 and 2. `SharedObj` is a global variable. `prvObj` is a thread-local variable.

between the shared and private copies of an object, only the modified groups are copied. To efficiently detect modifications, a version number is associated with each group of fields of each copy of the object. The algorithm works as follows: (1) read the shared object reference using LL; (2) copy data and version numbers in all modified groups of fields of the currently shared copy of the object into the corresponding groups of fields of the current thread's private copy; (3) do the computation on the private copy, updating fields in some group, and incrementing the corresponding version number; (4) switch the references between the shared object and the private object using SC. The algorithm is more complicated than Herlihy's algorithm for small objects in Figure 3 mainly because of the loop over groups of fields, the conditional behavior depending on which groups of fields changed, and the use of version numbers to efficiently detect changes.

Our analysis cannot directly show that the algorithm is atomic, due to the use of version numbers. Our analysis algorithm is able to show that a version of the algorithm that does not use version numbers is atomic. We then show that the transformations that optimize the algorithm by introducing and using version numbers preserve atomicity; this is relatively easy.

We show that the non-blocking algorithm for large objects (specifically, Algorithm 3 in Figure 9) is atomic. The procedure call `copy(prvObj.data[i],m.data[i])` copies the data in `m.data[i]` to `prvObj.data[i]`. Lines 3-8 are an unrolled loop that copies each group of fields; we unrolled the loop because our current analysis does not support nested loops, although it could be extended to do so. The procedure call `compute(prvObj,g)` does computation based on the data in `prvObj` and writes the result into `prvObj.data[g]`.

Algorithm 3 in Figure 9 differs in some minor ways from the original algorithm in [GH04]. It does not contain the redundant array `old` used in [GH04]. Like Herlihy's algorithm in Figure 3, it uses VL (line 6) to prevent errors due to inconsistent states of `prvObj` that may result from updates during copying (line 5). [GH04] simply assumed that such errors do not occur. Also, we omit the guard predicate used in [GH04], which is used to optimize cases where `compute` is applied in a state in which it performs no updates.

Algorithm 1 in Figure 8 is a simplified version of the algorithm, in which all data of the shared object (*i.e.*, `m`) are copied into the working object (*i.e.*, `prvObj`) of the current thread in every iteration of the loop. Like `prv` in Herlihy's algorithm in Figure 3, `prvObj` is a quasi-unique local reference. Moreover, for all `i`, `prvObj.data[i]` is dead at the end of the loop's body under all normal terminations. Therefore, the loop is pure. By the same reasoning as for the non-blocking algorithm for small objects in Figure 3, the procedure of Algorithm 1 in Figure 8 is atomic.

Algorithm 2 in Figure 8 is an improved version of Algorithm 1 in which the copy is omitted from `m.data[i]` to `prvObj.data[i]` when those two locations already contain the same value. Algorithm 2 clearly has the same behavior as Algorithm-1. Therefore, the procedure in Algorithm 2 is atomic.

```
proc alg3(SharedObj, g)
01  loop
02    local m = LL(SharedObj) in
03      local newVersion[1] = m.version[1] in
04        if (newVersion[1] != prvObj.version[1])
05          copy(prvObj.data[1], m.data[1]);
06          if (!VL(SharedObj)) continue;
07          prvObj.version[1] = newVersion[1];
08      ... // repeat lines 3-7 W times, incrementing the index each time, to copy each group of fields.
09      compute(prvObj, g);
10      prvObj.version[g]++;
11      if (SC(SharedObj, prvObj))
12        prvObj = m;
13        return;
14      else
15        prvObj.version[g] = 0;
```

Figure 9: Gao and Hesselink's non-blocking algorithm for large objects: Algorithm 3. `SharedObj` is a global variable. `prvObj` is a thread-local variable.

Algorithm 3 in Figure 9 is an improved version of Algorithm 2 in which version numbers are used to efficiently and conservatively check whether `m.data[i]` and `prvObj.data[i]` are equal. "Conservatively" here means that the check might return false when they contain the same value (*e.g.*, because the values stored in `m.data[i]` and `prvObj.data[i]` happen to be equal), but this merely causes the code in the full algorithm to do an unnecessary copy (*i.e.*, the copy does not actually change the value of `prvObj.data[i]`). The last statement "`prvObj.version[g] = 0`" is needed so that the update to `prvObj.version[g]` from line 10 will be discarded if the SC fails. Algorithm 3 clearly has the same behaviors as Algorithm 2. Therefore, the procedure in Algorithm 3 is atomic.

To evaluate the benefit of our atomicity analysis compared to a traditional partial-order reduction, we implemented Algorithm 3 in the model checker SPIN [Hol03]. We wrote a driver with 3 threads that concurrently invoke arithmetic operations on a shared object with 3 integer fields, each in its own group. The numbers of reachable states are: 4,069,080 with no optimization; 452,043 with SPIN's built-in partial-order reduction; 69,215 with the procedure body declared as atomic, as inferred by our analysis algorithm; and 4619 with both optimizations.

6.3 DCAS-Based Double-ended Queue

6.3.1 The DCAS operation

There is a growing realization that the synchronization operations on single memory locations, such as CAS, are not expressive enough to support designing efficient non-blocking algorithm [ADF$^+$00]. Double compare-and-swap (DCAS) shown in Figure 10 is a proposed stronger synchronization operation. DCAS is similar as CAS; it just adds another address `addr2` which is handled together with `addr1`. It is easy to give theorems for DCAS analogous to Theorems 5.3, 5.4, and 5.5 for LL/SC.

6.3.2 The Array-based Double-ended Queue

Agesen et al. proposed an array-based double-ended queue [ADF$^+$00] using DCAS as shown in Figure 11. The algorithm shown in Figure 11 is slightly simpler than the algorithm in [ADF$^+$00], because we removed an optional optimization from each procedure. In a double-ended queue (DEQ), elements can be enqueued (pushed) or dequeued (popped) from both ends. The array-based DEQ is bounded: it can hold at most `lengthS` elements. DEQ has four procedures: `popRight`, `pushRight`, `popLeft`, and `pushLeft`. We show only `popRight` and `pushRight`. The other two are similar.

```
boolean DCAS(val *addr1, val *addr2, val old1, val old2, val new1, val new2){
  atomically{
    if ((*addr1 == old1) && (*addr2 == old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else
      return false
  }
}
```

Figure 10: The double compare-and-swap operation shown in C language format.

All elements are saved in an array `S`. Global variable `R` contains the index of the next position to insert an element on the right. Thus, `R-1` is the position which holds the element to pop from the right.

The procedure `popRight` accesses two shared variables `R` and `S[newR]`. The loop in `popRight` is pure, because in all normal iterations, there is no update on shared variables, and all local variables are dead at the end of the loop body. `popRight` has two exceptional variants. One exceptional variant is from line 3 to line 9, in which the first DCAS returns true. According to our theorem for DCAS (the analogue of Theorem 5.4 for LL/SC), there is no write by other threads to shared variable `R` between line 3 and line 8, and there is no write to shared variable `S[newR]` by other threads between line 5 and line 8. All other accesses between line 3 and line 8 are to local variables. Thus, all the actions in this variant have atomicity type B, so this exceptional variant is atomic. The other exceptional variant is from line 3 to line 12. It is also atomic. The analysis is similar to the analysis for the first variant.

The procedure `pushRight` accesses two shared variables, `R` and `S[oldR]`. The loop in `popRight` is pure. `popRight` has two exceptional variants. One is from line 2 to line 8. Our analysis shows that other threads cannot update the shared variables `R` and `S[oldR]` during execution of this variant, and that it is atomic. Similarly, the exceptional variant from line 2 to line 11 is atomic.

Therefore, according to Theorem 5.2, the procedures `pushRight` and `popRight` are atomic.

7 Conclusions

This paper presents a static analysis to infer atomicity of code blocks in programs with non-blocking synchronization and applies it to four case studies. The concept of pure loop used in our analysis formally expresses a common design pattern for non-blocking algorithms. Although we need to slightly modify some of the case studies before applying our analysis, we consider the results encouraging, since we do not know of any other algorithmic (*i.e.*, automatable) analysis that can show atomicity of the same (or larger) code blocks in the modified or original versions. Our analysis significantly reduces the number of states considered during subsequent analysis, such as verification.

Acknowledgment

We thank Cormac Flanagan for helpful comments and Eran Yahav for help with TVLA.

References

[ADF$^+$00] Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. In *SPAA '00: Proceedings*

```
// Initially L == 0, R == 1; S[0..lengthS-1] of "null"
proc popRight()
  local newS = null in
    loop
      local oldR = R in
      local newR = (oldR - 1) mod lengthS in
      local oldS = S[newR]in
        if (oldS == null)
          if (oldR == R)
            if (DCAS(&R, &S[newR], oldR, oldS, oldR, oldS))
              return "empty";
        else
          if (DCAS(&R, &S[newR], oldR, oldS, newR, newS))
            return oldS;

proc pushRight(v)
  loop
    local oldR = R in
    local newR = (oldR + 1) mod lengthS in
    local oldS = S[oldR] in
      if (oldS != null)
        if (oldR == R)
          if (DCAS(&R, &S[oldR], oldR, oldS, oldR, oldS))
            return "full";
      else
        if (DCAS(&R, &S[oldR], oldR, oldS, newR, v))
          return "okay";
```

Figure 11: The array-based double-ended queue using DCAS. Variable R and array S are shared.

of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, pages 137–146. ACM Press, 2000.

[FF04] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. of ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–267. ACM Press, 2004.

[FFQ05] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4), April 2005.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[GH04] Hui Gao and Wim H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 44–56, 2004.

[Her93] Maurice P. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[HW90] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[Mic04a] Maged M. Michael. Private communication, 2004.

[Mic04b] Maged M. Michael. Scalable lock-free dynamic memeory allocation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* ACM Press, June 2004.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceeding of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275. ACM Press, 1996.

[Pel98] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[QRR04] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *Proc. 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–255. ACM Press, 2004.

[WS03] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. In *Third Workshop on Runtime Verification (RV03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2003.

[WS05] Liqiang Wang and Scott D. Stoller. Static analysis for programs with non-blocking synchronization. In *Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM Press, June 2005.

[WS06a] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM Press, 2006.

[WS06b] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, February 2006.

[YS03] Eran Yahav and Mooly Sagiv. Automatically verifying concurrent queue algorithms. In *Proc. Workshop on Software Model Checking (SoftMC'03)*, volume 89(3) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2003.

Appendix

A Simulation of NFQ$'$

By construction, NFQ$'$ is more non-deterministic than NFQ and can simulate all behaviors of NFQ. We show below that linearizability of NFQ with respect to any specification (of the kind defined in [HW90]) follows from linearizability of NFQ$'$ with respect to that specification augmented freely with calls to `UpdateTail`.

Following [HW90], a specification *Spec* is a prefix-closed set of single-object sequential histories.

Based on a given *Spec* for NFQ, the specification *Spec*$'$ for NFQ$'$ is defined by introducing a new thread P_{Tail} that executes the `UpdateTail` procedure: for each $H \in \mathit{Spec}$, for each well-formed sequential history H' that can be obtained by inserting $\langle Q.\mathit{UpdateTail}(), P_{Tail}\rangle$ (which denotes an invocation of `Q.UpdateTail` by P_{Tail}) and $\langle Q.OK(), P_{Tail}\rangle$ (which denotes a return, also called response, from `Q.UpdateTail` by P_{Tail}) in H, add H' to *Spec*$'$.

Recall that $complete(H_r)$ is the subsequence of H_r obtained by deleting the invocations without matching responses. Recall that H is linearizable with respect to *Spec* if H can be extended to some history H_r by adding response events, such that

*L*1. $complete(H_r)$ is equivalent to some $S \in Spec$, *i.e.*, $\forall$ thread $P : complete(H_r)|P = S|P$, and

*L*2. $<_H \subseteq <_S$

Theorem A.1. *If NFQ′ is linearizable with respect to Spec′ then NFQ is linearizable with respect to Spec.*

Proof. Let σ be an execution of NFQ. Let H be the corresponding history of NFQ obtained by deleting all actions except for call/return. Construct an execution σ' from σ as follows.

- Replace each successful SC(Tail,_) with an execution of `UpdateTail()` by P_{Tail}. Success of the original SC implies that the SC in `UpdateTail()` succeeds.
- Delete each unsuccessful execution of SC(Tail,_).

One can show that σ' is an execution of NFQ′. Let H' denote the corresponding history. Linearizability of NFQ′ with respect to *Spec*′ implies that there is an execution H'_r of H' and a sequential history $S' \in Spec'$ such that

*L*1′. $complete(H'_r)$ is equivalent to S', and

*L*2′. $<_{H'} \subseteq <_{S'}$

Let H_r and S be the subsequences of H'_r and S', respectively, obtained by deleting all invocations of `UpdateTail()` and the matching responses. Note that H_r is an extension of H by adding response events. Also, $S \in Spec$, by design of *Spec*′. Note that $complete(H_r)$ is equivalent to S; this follows from *L*1′, and the fact that P_{Tail} does not appear in H_r or S, so the projection of both onto P_{Tail} is the empty sequence.

Let $f(<)$ denote the projection of an ordering $<$ onto operations of all threads other than P_{Tail}. *L*2′ implies $f(<_{H'}) \subseteq f(<_{S'})$. Note that $<_H = f(<_{H'})$ and $<_S = f(<_{S'})$. So $<_H \subseteq <_S$. ☐

The converse of the above theorem can be proved similarly. Therefore, this approach, *i.e.*, proving the linearizability of NFQ by showing the linearizability of NFQ′, is complete.

B Semantics of SYNL

B.1 Domains

The semantic domains used in the semantics of SYNL are shown in Figure 12. *GVar* and *LVar* are the sets of global and local variables, respectively. $A \rightharpoonup B$ is the type of partial functions from A to B. A program state is a tuple $\langle G, H, T\rangle$ containing a global store G, a heap H, and a sequence T containing, for each thread, a local store L and a statement to be executed next. The address of a record structure (an object or array) is often stored in a reference variable. To access the record structure, there are two maps: the first map is from reference variable to address, and has type *GStore* or *LStore*; the second map is from address to structure, and has type *Heap*. $H[p \mapsto d]$ denotes a new heap that is identical to H except it maps address p to record d.

The *Struct* domain allows arrays with gaps in the set of legal indices. This generality is unnecessary but harmless; our proofs remain valid for semantic domains that exclude arrays with gaps.

The semantics of LL/VL/SC associates a set of thread identifiers with each global variable, each field of each object, and each element of each array. We call this information *synchronization state* and represent it using the *SyncState* domain. The set contains the identifiers of threads whose most recent LL on that variable is still valid, *i.e.*, the variable has not been updated (by a SC) since then. A LL(v) by thread i adds i to the set associated with v. A successful SC(v, val) empties the set.

$\rightarrow_i$ is the transition relation of thread i. $\rightarrow$ is the transition relation of the program.

$$\begin{array}{rcl}
\varphi & \in & State = GStore \times Heap \times Thread^* \\
H & \in & Heap = Addr \rightharpoonup Struct \\
d & \in & Struct = (Field \rightharpoonup Val \times SyncState) \cup (Index \rightharpoonup Val \times SyncState) \\
L & \in & LStore = LVar \rightharpoonup Val \\
t & \in & Thread = LStore \times Statement \\
v & \in & Val = Addr \cup int \cup bool \\
i & \in & TID = Nat \\
G & \in & GStore = GVar \rightharpoonup Val \times SyncState \\
Y & \in & SyncState = Set(TID) \\
idx & \in & Index = Nat \\
p & \in & Addr \\
T & \in & Thread^* \\
\rightarrow_i & \subseteq & State \times State \\
\rightarrow & \subseteq & State \times State
\end{array}$$

Figure 12: Semantic domains for SYNL.

$$\begin{array}{rcl}
Expr & ::= & p \\
Statement & ::= & \texttt{inloop}\ s\ s \mid \texttt{done} \mid \texttt{inlocal}\ x\ s \mid \texttt{insync}\ p\ s \\
E & ::= & [\,] \mid E.fd \mid E[e] \mid x[E] \mid prim(e_1, ..., e_n, E, v_1, ..., v_m) \mid \mathrm{SC}(loc,\ E) \\
 & & \mid \mathrm{CAS}(loc,E,e) \mid \mathrm{CAS}(loc,v,E) \mid loc := E \mid \texttt{if}\ E\ s\ s \mid \texttt{loop}\ E \\
 & & \mid \texttt{inloop}\ s\ E \mid E;s \mid \texttt{local}\ x := E\ \texttt{in}\ s \mid \texttt{inlocal}\ x\ E \\
 & & \mid \texttt{return}\ E \mid \texttt{synchronized}\ E\ s \mid \texttt{insync}\ p\ E
\end{array}$$

Figure 13: Evaluation contexts of SYNL.

B.2 Evaluation Contexts

The evaluation contexts of SYNL are defined in Figure 13. Evaluation contexts are used to identify the next part of an expression or statement to be evaluated. An evaluation context E is an expression or statement with a hole in place of the next sub-expression or sub-statement to be evaluated. Expressions evaluate to expressions and eventually become values. Statements evaluate to statements and eventually become the `done` statement or get blocked or stuck.

Figure 13 also introduces additional expression and statement forms that help keep track of computations. The `inlocal` statement denotes that execution is proceeding inside a `local` statement. The `inloop` and `insync` statements are similar.

Let $T[i]$ denote the i^{th} element of sequence T starting with 1. In a state $\langle G, H, T\rangle$, where $T[i]$ contains `insync` p, we say that thread i holds lock p. In a state where no thread holds lock p, we say that lock p is free. We refer to this as the state of the lock.

Note that the grammar for evaluation contexts does not contain a production like LL(E) or VL(E); if it did, for example, LL(x) would evaluate to, *e.g.*, LL(3), if the value of x is 3.

B.3 Transition Rules

Let π_i select the i^{th} component of a tuple. For example, $\pi_2(\langle a, b, c\rangle) = b$. For a mapping L, let $L - x$ denote L with x removed from its domain.

The transition rules of SYNL are shown in Figures 14 and 15, and

$$val(x, G, L) = \begin{cases} \pi_1(G(x)) & \text{if } x \in dom(G) \\ \pi_1(L(x)) & \text{if } x \in dom(L) \\ \bot & \text{otherwise} \end{cases}$$

$$
\begin{array}{rcl}
G, H, T.\langle L, E[x]\rangle.T' & \to_i & G, H, T.\langle L, E[v]\rangle.T', \\
 & & \text{if } v = \mathit{val}(x, G, L) \wedge v \neq \bot \\
G, H, T.\langle L, E[p.\mathit{fd}]\rangle.T' & \to_i & G, H, T.\langle L, E[\pi_1(H(p)(\mathit{fd}))]\rangle.T', \\
 & & \text{if } p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
G, H, T.\langle L, E[p[\mathit{idx}]]\rangle.T' & \to_i & G, H, T.\langle L, E[\pi_1(H(p)(\mathit{idx}))]\rangle.T', \\
 & & \text{if } p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
G, H, T.\langle L, E[\texttt{new}\ C]\rangle.T' & \to_i & G, H[p \mapsto d], T.\langle L, E[p]\rangle.T', \text{ where } p \notin \mathit{dom}(H), \\
 & & \text{Note: } d \text{ is a record of type C, and appropriately initialized} \\
G, H, T.\langle L, E[\mathit{prim}(\bar{v})]\rangle.T' & \to_i & G, H, T.\langle L, E[v_0]\rangle.T', \text{ where } v_0 = [\![\mathit{prim}]\!](\bar{v}) \\
 & & \text{Note: primitive operations have no side effect} \\
G, H, T.\langle L, E[\texttt{LL}(x)]\rangle.T' & \to_i & G[x \mapsto \langle v, Y \cup \{i\}\rangle], H, T.\langle L, E[v]\rangle.T', \\
 & & \text{if } x \in \mathit{dom}(G) \wedge \langle v, Y\rangle = G(x) \\
G, H, T.\langle L, E[\texttt{LL}(x.\mathit{fd})]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{fd} \mapsto \langle v, Y \cup \{i\}\rangle]], T.\langle L, E[v]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \langle v, Y\rangle = H(p)(\mathit{fd}) \\
 & & \text{Note: } p \in \mathit{dom}(H) \text{ implies } p \neq \bot \\
G, H, T.\langle L, E[\texttt{LL}(x[\mathit{idx}])]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{idx} \mapsto \langle v, Y \cup \{i\}\rangle]], T.\langle L, E[v]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \langle v, Y\rangle = H(p)(\mathit{idx}) \\
G, H, T.\langle L, E[\texttt{VL}(x)]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{true}]\rangle.T', \text{ if } x \in \mathit{dom}(G) \wedge i \in \pi_2(G(x)) \\
G, H, T.\langle L, E[\texttt{VL}(x)]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T', \text{ if } x \in \mathit{dom}(G) \wedge i \notin \pi_2(G(x)) \\
G, H, T.\langle L, E[\texttt{VL}(x.\mathit{fd})]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{fd} \in \mathit{dom}(H(p)) \wedge i \in \pi_2(H(p)(\mathit{fd})) \\
G, H, T.\langle L, E[\texttt{VL}(x.\mathit{fd})]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{fd} \in \mathit{dom}(H(p)) \wedge i \notin \pi_2(H(p)(\mathit{fd})) \\
G, H, T.\langle L, E[\texttt{VL}(x[\mathit{idx}])]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{idx} \in \mathit{dom}(H(p)) \wedge i \in \pi_2(H(p)(\mathit{idx})) \\
G, H, T.\langle L, E[\texttt{VL}(x[\mathit{idx}])]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{idx} \in \mathit{dom}(H(p)) \wedge i \notin \pi_2(H(p)(\mathit{idx})) \\
G, H, T.\langle L, E[\texttt{SC}(x, v)]\rangle.T' & \to_i & G[x \mapsto \langle v, \emptyset\rangle], H, T.\langle L, E[\texttt{true}]\rangle.T', \\
 & & \text{if } x \in \mathit{dom}(G) \wedge i \in \pi_2(G(x)) \\
G, H, T.\langle L, E[\texttt{SC}(x, v)]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T', \text{ if } x \in \mathit{dom}(G) \wedge i \notin \pi_2(G(x)) \\
G, H, T.\langle L, E[\texttt{SC}(x.\mathit{fd}, v)]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{fd} \mapsto \langle v, \emptyset\rangle]], T.\langle L, E[\texttt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{fd} \in \mathit{dom}(H(p)) \wedge i \in \pi_2(H(p)(\mathit{fd})) \\
G, H, T.\langle L, E[\texttt{SC}(x.\mathit{fd}, v)]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge i \notin \pi_2(H(p)(\mathit{fd})) \\
G, H, T.\langle L, E[\texttt{SC}(x[\mathit{idx}], v)]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{idx} \mapsto \langle v, \emptyset\rangle]], T.\langle L, E[\texttt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \\
 & & \quad \wedge \mathit{idx} \in \mathit{dom}(H(p)) \wedge i \in \pi_2(H(p)(\mathit{idx})) \\
G, H, T.\langle L, E[\texttt{SC}(x[\mathit{idx}], v)]\rangle.T' & \to_i & G, H, T.\langle L, E[\texttt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge i \notin \pi_2(H(p)(\mathit{idx}))
\end{array}
$$

Figure 14: Transition rules of SYNL, part 1. Here, $i = |T| + 1$.

The transition rule $G, H, T \to_i G, H, T.\langle L, s\rangle$ in Figure 15 models the environment calling a procedure in a new thread. Recall that SYNL allows procedures to be called concurrently by the environment.

The rule $G, H, T.\langle L, \texttt{done}\rangle.T' \to_i G, H, T.\langle L', s\rangle.T'$ in Figure 15 models the environment calling a procedure in an existing thread that finished its previous procedure calls.

$$
\begin{array}{rcl}
G, H, T.\langle L, E[\mathtt{CAS}(x, v_1, v_2)]\rangle.T' & \to_i & G[x \mapsto \langle v_2, \pi_2(G(x))\rangle], H, T.\langle L, E[\mathtt{true}]\rangle.T', \\
 & & \text{if } x \in \mathit{dom}(G) \wedge \pi_1(G(x)) = v_1 \\
G, H, T.\langle L, E[\mathtt{CAS}(x, v_1, v_2)]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{false}]\rangle.T', \\
 & & \text{if } x \in \mathit{dom}(G) \wedge \pi_1(G(x)) \neq v_2 \\
G, H, T.\langle L, E[\mathtt{CAS}(x.\mathit{fd}, v_1, v_2)]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{fd} \mapsto \langle v_2, \pi_2(H(p)(\mathit{fd}))\rangle]], T.\langle L, E[\mathtt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \pi_1(H(p)(\mathit{fd})) = v_1 \\
G, H, T.\langle L, E[\mathtt{CAS}(x.\mathit{fd}, v_1, v_2)]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \pi_1(H(p)(\mathit{fd})) \neq v_1 \\
G, H, T.\langle L, E[\mathtt{CAS}(x[\mathit{idx}], v_1, v_2)]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{idx} \mapsto \langle v_2, \pi_2(H(p)(\mathit{fd}))\rangle]], T.\langle L, E[\mathtt{true}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \pi_1(H(p)(\mathit{idx})) = v_1 \\
G, H, T.\langle L, E[\mathtt{CAS}(x[\mathit{idx}], v_1, v_2)]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{false}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
 & & \quad \wedge \pi_1((H(p)(\mathit{idx})) \neq v_1 \\
G, H, T.\langle L, E[x := v]\rangle.T' & \to_i & G', H, T.\langle L', E[\mathtt{done}]\rangle.T' \\
 & & \text{if } (x \in \mathit{dom}(G) \wedge G' = G[x \mapsto \langle v, \pi_2(G(x))\rangle] \wedge L' = L) \\
 & & \quad \vee\ (x \in \mathit{dom}(L) \wedge L' = L[x \mapsto v] \wedge G' = G) \\
G, H, T.\langle L, E[x.\mathit{fd} := v]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{fd} \mapsto \langle v, \pi_2(H(p)(\mathit{fd}))\rangle]], T.\langle L, E[\mathtt{done}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{fd} \in \mathit{dom}(H(p)) \\
G, H, T.\langle L, E[x[\mathit{idx}] := v]\rangle.T' & \to_i & G, H[p \mapsto H(p)[\mathit{idx} \mapsto \langle v, \pi_2(H(p)(\mathit{idx}))\rangle]], T.\langle L, E[\mathtt{done}]\rangle.T' \\
 & & \text{if } p = \mathit{val}(x, G, L) \wedge p \in \mathit{dom}(H) \wedge \mathit{idx} \in \mathit{dom}(H(p)) \\
G, H, T.\langle L, E[\mathtt{if\ true}\ s_1\ s_2]\rangle.T' & \to_i & G, H, T.\langle L, E[s_1]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{if\ false}\ s_1\ s_2]\rangle.T' & \to_i & G, H, T.\langle L, E[s_2]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{loop}\ s]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{inloop}\ s\ s]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{inloop}\ s\ E'[\mathtt{continue}]]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{inloop}\ s\ s]\rangle.T', \\
G, H, T.\langle L, E[\mathtt{inloop}\ s\ E'[\mathtt{break}]]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{done}]\rangle.T', \\
 & & \text{if } E' \text{ does not contain } \mathtt{inloop} \\
G, H, T.\langle L, E[\mathtt{inloop}\ s\ \mathtt{done}]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{inloop}\ s\ s]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{synchronized}\ p\ s]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{insync}\ p\ s]\rangle.T', \\
 & & \text{if } T \text{ and } T' \text{ do not contain } \mathtt{insync}\ p \\
G, H, T.\langle L, E[\mathtt{insync}\ p\ \mathtt{done}]\rangle.T' & \to_i & G, H, T.\langle L, E[\mathtt{done}]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{done}; s]\rangle.T' & \to_i & G, H, T.\langle L, E[s]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{local}\ x = v\ \mathtt{in}\ s]\rangle.T' & \to_i & G, H, T.\langle L[x \mapsto v], E[\mathtt{inlocal}\ x\ s]\rangle.T', \text{ if } x \notin \mathit{dom}(L) \\
G, H, T.\langle L, E[\mathtt{inlocal}\ x\ \mathtt{done}]\rangle.T' & \to_i & G, H, T.\langle L - x, E[\mathtt{done}]\rangle.T' \\
G, H, T.\langle L, E[\mathtt{return}]\rangle.T' & \to_i & G, H, T.\langle L, \mathtt{done}\rangle.T' \\
G, H, T.\langle L, E[\mathtt{return}\ v]\rangle.T' & \to_i & G, H, T.\langle L, \mathtt{done}\rangle.T' \\
G, H, T & \to_i & G, H, T.\langle L, s\rangle, \text{ where the program declares a procedure} \\
 & & \qquad\qquad p(\bar{x})\ \{s\}, \text{ and } \mathit{dom}(L) = \bar{x}. \\
G, H, T.\langle L, \mathtt{done}\rangle.T' & \to_i & G, H, T.\langle L', s\rangle.T', \text{ where the program declares a procedure} \\
 & & \qquad\qquad p(\bar{x})\ \{s\}, \text{ and } \mathit{dom}(L') = \bar{x}.
\end{array}
$$

Figure 15: Transition rules of SYNL, part 2. Here, $i = |T| + 1$.

Note that the only actions performed by the environment are calls to procedures defined in the program.

This semantics does not model garbage collection of unreachable structures or terminated threads. This semantics allows the heap, procedure arguments, *etc.*, to contain invalid addresses, *i.e.*, addresses not in $\mathit{dom}(H)$. Attempting to dereference them causes the thread to get stuck. We can consider `null` to be such an invalid address, if getting stuck is appropriate semantics for attempting to dereference `null`. Otherwise, we could introduce a special `null` value in *Addr*, and add appropriate transition rules for dereferencing `null`. The semantics for `return` does not explicitly model the communication of the return value to the environment; it could easily be modified to do so.

C Proof of Theorem 4.1

Let $\sigma = G_0, H_0, T_0 \rightarrow_{t_0} G_1, H_1, T_1 \rightarrow_{t_1} \ldots$ be an execution of a program P.

Let $[i..j]$ denote the set of integers from i to j. Let $max(m : p(m))$ denote the largest value of m that satisfies predicate $p(m)$. Let I be the indices in σ of all transitions that are part of normal iterations of pure loops. Let σ' be the execution constructed from σ by deleting transitions in I. Deleting transitions involves adjusting the states as follows. The j^{th} state in σ' corresponds to the $f(j)^{th}$ state in σ, where $f(j) = max(m : |[0..m-1] \backslash I| = j)$, *i.e.*, $f(j)$ is the maximal m such that there are j transitions remaining from the 0^{th} to $(m-1)^{th}$ transitions after deleting transitions in I. Let τ_j denote the j^{th} transition in σ, *i.e.*, G_j, H_j, $T_j \rightarrow_i G_{j+1}$, H_{j+1}, T_{j+1}. Let τ'_j denote the j^{th} transition of σ', *i.e.*, G'_j, H'_j, $T'_j \rightarrow_i G'_{j+1}$, H'_{j+1}, T'_{j+1}.

The j^{th} state in σ is denoted as G_j, H_j, T_j, and the components of $T_j[k]$ are denoted as $L_j[k]$ and $S_j[k]$. The j^{th} state in σ' is denoted as G'_j, H'_j, T'_j and is computed as follows. Let $p \in Addr$, $fd \in Field$, and $x \in LVar \cup GVar$. The treatment of arrays is very similar to the treatment of records, so for brevity, we show only the latter.

Let $WriteH(\sigma, p, fd)$ denote the indices of transitions in σ that update $H(p)(fd)$. For global variable x, let $WriteG(\sigma, x)$ denotes the indices of transitions in σ that update $G(x)$. Note that a LL is considered as a write, since it updates the synchronization state of x, which is part of $G(x)$. Failed SC and CAS transitions are not considered as writes.

Let $reachingDefH(\sigma, j, p, fd)$ be the index in σ of the last transition that occurs before the j^{th} state and updates $H(p)(fd)$, *i.e.*, $reachingDefH(\sigma, j, p, fd) = max(WriteH(\sigma, p, fd) \cap [0..j-1])$. Note that the value written by that transition is the value seen in the state immediately after it. Let $reachingDefH'(\sigma, j, p, fd)$ be defined in the same way but ignoring updates in normal iterations of pure loops, *i.e.*,$reachingDefH'(\sigma, j, p, fd)$ $= max(WriteH(\sigma, p, fd) \backslash I \cap [0..j-1])$. Define $reachingDefG(\sigma, j, x)$ and $reachingDefG'(\sigma, j, x)$ similarly.

$$\begin{aligned} H'_j(p)(fd) &= \texttt{let } k = reachingDefH'(\sigma, f(j), p, fd) \\ &\quad\ \texttt{in } H_{k+1}(p)(fd) \\ G'_j(x) &= \texttt{let } k = reachingDefG'(\sigma, f(j), x) \\ &\quad\ \texttt{in } G_{k+1}(x) \end{aligned}$$

Let $Trans(\sigma, i)$ denote the indices of transitions of thread i in σ. $WriteL_i$, $reachingDefL_i$, and $reachingDefL'_i$ are analogous to $WriteG$, $reachingDefG$, and $reachingDefG'$, respectively, except they are for local variables of thread i.

$$\begin{aligned} L'_j[i](x) &= \texttt{let } k = reachingDefL'_i(\sigma, f(j), x) \\ &\quad\ \texttt{in } L_{k+1}(x) \\ S'_j[i] &= \texttt{let } k = max((Trans(\sigma, i) \backslash I) \cap [0..f(j)-1]) \\ &\quad\ \texttt{in } S_{k+1}[i] \\ T'_j &= \langle L'_j, S'_j \rangle \end{aligned}$$

The following formulas for σ express the fact that each variable contains the value most recently written to it.

$$\begin{aligned} H_j(p)(fd) &= \texttt{let } k = reachingDefH(\sigma, i, p, fd) \\ &\quad\ \texttt{in } H_{k+1}(p)(fd) \\ G_j(x) &= \texttt{let } k = reachingDefG(\sigma, j, x) \\ &\quad\ \texttt{in } G_{k+1}(x) \\ L_j[i](x) &= \texttt{let } k = reachingDefL_i(\sigma, j, x)) \\ &\quad\ \texttt{in } L_{k+1}(x) \\ S_j[i] &= \texttt{let } k = max(Trans(\sigma, i) \cap [0..j-1]) \\ &\quad\ \texttt{in } S_{k+1}[i] \end{aligned}$$

A storage location is a local variable, global variable, field, or array element.

An update by τ_j to a storage location is *visible* to a transition τ_k if $reachingDefH(\sigma, k, p, fd) = j$, $reachingDefG(\sigma, k, x) = j$, or $reachingDefL_i(\sigma, k, x) = j$, depending on the kind of storage location updated by τ_j. An update by τ_j to synchronization state is *visible* to τ_k if (a) τ_j is LL and τ_k is the matching SC by the same thread, or (b) τ_j is a lock acquire and τ_k is the corresponding lock release by the same thread. The visibility of successful SC is not defined; it is not needed in this paper, because a successful SC cannot be a pure action. Note that VL, failed SC, and CAS operations do not update synchronization state.

Lemma C.1. *An update by a transition in I is visible only to transitions in I.*

Proof. We suppose an update by τ_j with $j \in I$ is visible to a transition τ_k with $k \notin I$, and we show a contradiction.

We consider cases corresponding to the kind of update.

Case 1. The update is a write to a storage location x. We consider cases based on the kind of storage location x.

Case 1.1: x is a global variable. This is impossible, because a pure action cannot update a global variable.

Case 1.2: x is a local variable or a field of an unshared object. $k \notin I$ implies that x is not loop-local, so condition (*1.iii*) in the definition of pure action implies x is accessed in every exceptional iteration of the loop (in which τ_j occurs). The normal iteration in which τ_j occurs must be followed (possibly after more normal iterations) by an exceptional iteration. Condition (*1.ii*) implies the first access to x in that exceptional iteration must be a write. Let j_2 be the index of that write. τ_{j_2} precedes τ_k, because τ_{j_2} is the first access to x after the sequence of normal iterations containing τ_j. The update to x by τ_{j_2} is visible to τ_k, so the update to x by τ_j is not, this is a contradiction.

Case 1.3: x is a field of an object o accessed by τ_j through a unique local reference variable v.

Case 1.3.1: $thread(\tau_j) = thread(\tau_k)$. The reasoning is the same as in case 1.2.

Case 1.3.2: $thread(\tau_j) \neq thread(\tau_k)$. By the same reasoning as in case 1.2, the sequence of normal iterations containing τ_j is followed by an exceptional iteration that contains a write τ_{j_2} to x. Furthermore, condition (*1.ii*) in the definition of pure action implies that τ_{j_2} is performed by dereferencing a quasi-unique local reference.

Since τ_j uses a unique local reference, some subsequent transition τ_o of $thread(\tau_j)$ must make a reference to o accessible to $thread(\tau_k)$ by updating a shared variable. Note that τ_o must precede τ_k. A pure action cannot update a shared variable, so τ_o must occur in or after the exceptional iteration in which τ_{j_2} occurs. If τ_o occurs in the exceptional iteration, τ_{j_2} precedes τ_o, because τ_{j_2} is performed by dereferencing a quasi-unique local reference to o, and $thread(\tau_j)$ does not have such a reference after τ_o. If τ_o occurs after the exceptional iteration, then clearly τ_{j_2} precedes τ_o. Thus, in both cases, by transitivity, τ_{j_2} precedes τ_k, so τ_k sees the update from τ_{j_2}, not the update from τ_j. This is a contradiction.

Case 1.4: x is a field of an object o accessed through a quasi-unique local reference variable v.

Case 1.4.1: $thread(\tau_j) = thread(\tau_k)$. The reasoning is the same as in case 1.2.

Case 1.4.2: $thread(\tau_j) \neq thread(\tau_k)$. The reasoning is almost the same as in case 1.3.2. Note that, although other threads (including $thread(\tau_k)$) may have references to o in locations accessed only in normal iterations of loops, these references cannot be stored into shared variables in the normal iterations. Therefore the reasoning in case 1.3.2 about a transition τ_o of $thread(\tau_j)$ that makes o accessible to $thread(\tau_k)$ when $thread(\tau_k)$ is outside of normal iterations is valid in this case as well.

Case 1.5: x is an array element. The analysis is similar to the analysis for a field access.

Case 2: The update is to synchronized state.

Case 2.1: τ_j performs a LL action.

Condition (*2.i*) in the definition of pure action implies that the LL in τ_j and the matching SC occur in the same normal iteration. An update by a LL action to synchronization state is visible only to the matching SC by the same thread. Therefore, the update by τ_j is not visible to τ_k. This is a contradiction.

Case 2.2: τ_j performs a SC action.

Condition (*2.ii*) implies that no successful SC can occur in I, and failed SC operations do not update the synchronization state, so τ_j does not perform any update visible to τ_k. This is a contradiction.

Case 2.3: `synchronized`.

`synchronized` statements are block-structured, so if some transition in I performs an acquire action, then I also contains the transition that performs the matching release action (which ends the `synchronized` block). Moreover, other threads cannot perform any operations on a lock while it is held by the current thread (an attempted acquire action blocks until the lock is freed). Hence, the updates to synchronization states by acquire and release transitions in I are visible only in I. $\square$

Lemma C.2. *For every transition τ'_j in σ',*
(i) τ'_j and $\tau_{f(j)}$ are transitions of the same thread, call it thread i, and
(ii) $S'_j[i] = S_{f(j)}[i]$, and
(iii) all locations read by τ'_j have the same value in state G'_j, H'_j, $L'_j[i]$ and state $G_{f(j)}$, $H_{f(j)}$, $L_{f(j)}[i]$.

Proof. Claims (i) and (ii) follow directly from the definitions of σ' and f. Claim (iii) follows immediately from Lemma C.1. $\square$

Lemma C.3. *σ' is an execution of the program P.*

Proof. A straightforward property of the operational semantics is that, if some transition rule shows that $\varphi_1 \rightarrow \varphi_2$ is a transition of P, and φ'_1 and φ'_2 are obtained from φ_1 and φ_2, respectively, by a change to some parts of the state that are not accessed by the transition according to Lemma C.2, then the same transition rule shows that $\varphi'_1 \rightarrow \varphi'_2$ is a transition of P. We conclude that τ'_j is a transition of the program based on the same transition rule used to show that $\tau_{f(j)}$ is a transition of the program. Therefore, σ' is an execution of the program P. $\square$

Lemma C.4. *σ and σ' contain the same states in which all threads are executing outside pure loops.*

Proof. All deleted transitions are in pure loops, so there is a one-to-one correspondence between states outside executions of pure loops in σ and states outside executions of pure loops in σ'. We show that the corresponding states are the same. By inspection of the formulas defining H', G', and T', deletion of a transition that updates a location x produces a difference between corresponding states in σ and σ' that propagates forward in σ' until it encounters either a transition that updates x or the end of x's scope. Condition (*1.ii*) in the definition of pure action implies that, for each such location x, at least one of these two things happens before the end of the pure loop. $\square$

Theorem 4.1 *Let σ be an execution of a program P. Suppose all loops in P are pure. Let σ' be an execution obtained from σ by deleting all transitions in all normal iterations of all pure loops in P. Then σ' is also an execution of P, and σ and σ' contain the same states in which all threads are executing outside pure loops.*
Proof. The theorem follows directly from Lemmas C.3 and C.4. $\square$