Logical Concurrency Control From Sequential Proofs

Jyotirmoy Deshmukh jyotirmoy@cerc.utexas.edu
G. Ramalingam grama@microsoft.com
Venkatesh Prasad Ranganath rvprasad@microsoft.com
Kapil Vaswani kapilv@microsoft.com

July 2009
Technical Report
MSR-TR-2009-81

In this paper, we consider the problem of making a sequential library safe for concurrent clients. Informally, given a sequential library that works satisfactorily when invoked by a sequential client, we wish to synthesize concurrency control code for the library that ensures that it will work satisfactorily even when invoked by a concurrent client (which may lead to overlapping executions of the library's procedures). Formally, we consider a sequential library annotated with assertions along with a proof that these assertions hold in a sequential execution. We show how such a proof can be used to derive a concurrency control for the library that guarantees that the library's execution will satisfy the same assertions even when invoked by a concurrent client. Secondly, we generalize this result by considering 2-state assertions that correspond to relations over a pair of program states. Such assertions can be used (as postconditions) to specify the desired functionality of procedures. Thus, the synthesized concurrency control ensures that procedures have the desired functionality even in a concurrent setting. Finally, we extend the approach to guarantee linearizability: any concurrent execution of a procedure is not only guaranteed to satisfy its specification, it also appears to take effect instantaneously at some point during its execution. A notable feature of our solution is that it is based on a *logical* notion of interference between threads: the derived concurrency control prevents threads from violating properties (by executing statements) that are to be preserved at a given program point, rather than preventing threads from accessing/modifying specific data.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Recent technology trends point to the increasing importance of concurrency. But even newly developed systems and programs will need to make heavy use of pre-existing libraries that are too valuable to be just discarded. Unfortunately, libraries that work perfectly well in a sequential setting may fail to work in a concurrent setting due to the use of imperative programming languages with mutable state in the form of global variables or heap-allocated data.

In this paper, we consider the problem of making a sequential library safe for concurrent clients. Informally, given a sequential library that works satisfactorily when invoked by a sequential client, we show how to synthesize concurrency control code for the library that ensures that it will work satisfactorily even when invoked by a concurrent client.

Consider the sequential library in Figure 1 (ignoring the `acquire` and `release` operations). The library consists of one procedure `Compute`, which applies an expensive function f to an input variable `num`. To avoid repeating the expensive computation on every invocation, the implementation caches the last input and the last result. If the current input matches the last input, the last computed result is returned instead.

This procedure works perfectly fine when used by a sequential client. However, if the procedure is used by a concurrent client, it is possible to have overlapping invocations of the procedure. In this situation, the procedure may return an incorrect answer. E.g., consider an invocation of "`Compute(5)`" subsequently followed by the concurrent invocations of "`Compute(5)`" and "`Compute(7)`". Assume that the second invocation of "`Compute(5)`" evaluates the condition in line 9, and proceeds to line 10. Assume a context switch occurs at this point, and the invocation of "`Compute(7)`" executes completely, overwriting `lastRes` in line 16. Now, when the invocation of "`Compute(5)`" resumes, it will erroneously return the (changed) value of `lastRes`.

Concurrency Control. Ensuring that programs function correctly in a concurrent setting requires the use of appropriate concurrency control mechanisms to prevent undesirable interleaving of different threads. However, augumenting a program with the right amount of synchronization that ensures both correctness and high performance is a challenging task.

In this paper, we address the problem of automatically making sequential code *thread-safe*: given some sequential code that works satisfactorily in a sequential setting, we wish to synthesize concurrency control that ensures that the given piece of code works correctly in a concurrent setting as well. Given the code in Figure 1, our approach is able to automatically synthesize the locking-based concurrency control mechanism (i.e. the `acquire` and `release` operations with the corresponding locks) shown in the figure. The reader may verify that this version of the library words correctly even in the presence of overlapping procedure invocations.

```
1 global int lastNum = 0, lastRes = f(0);
2 //@requires lastRes == f(lastNum)
3 //@ensures lastRes == f(lastNum)
4 //@ensures lastNum == num
5 //@returns f(num)
6 Compute(num)
7 {
8   acquire(l);
9   if (lastNum == num) {
10    res = lastRes;
11  } else {
12    release(l);
13    res = f(num);
14    acquire(l);
15    lastNum = num;
16    lastRes = res;
17  }
18  release(l);
19  return res;
20 }
```

Figure 1: A procedure in a library that applies a function f to an input variable num and caches the result for subsequent invocations. $f(x)$ represents a mathematical function, such as x^3, rather than an imperative procedure.

The Problem. To formalize our problem, we must first formalize the correctness criterion for a library: what does it mean to say that a library works correctly in a sequential or concurrent setting? We assume that the desired properties of the library are specified via a set of assertions. Further, we assume that library satisfies these assertions in a sequential setting: i.e., any possible execution of the library, with a sequential client, is assumed to satisfy the given assertions. Our goal is to ensure that any possible concurrent execution of the library also satisfies the given assertions.

For our running example in Figure. 1, lines 2-5 provide a specification for procedure `Compute`. Line 5 specifies the desired functionality of the procedure (i.e., `Compute` returns the value $f($`num`$)$), while lines 2-4 indicate the invariants about the library's own state that the procedure maintains. (In general, the interpretion of pre/post-conditions in a concurrent execution is complicated; we will later provide precise definitions of this interpretation.)

Logical Concurrency Control From Proofs. One of the key challenges in coming up with concurrency control is finding the *degree of isolation required between concurrently executing threads*: what interleavings between threads can be permitted? A very conservative solution may prevent more interleavings than necessary; hence, it can reduce the concurrency in the system. A very aggressive solution focused on enabling more concurrency may introduce subtle bugs.

The fundamental thesis explored in this paper is the following: a proof that

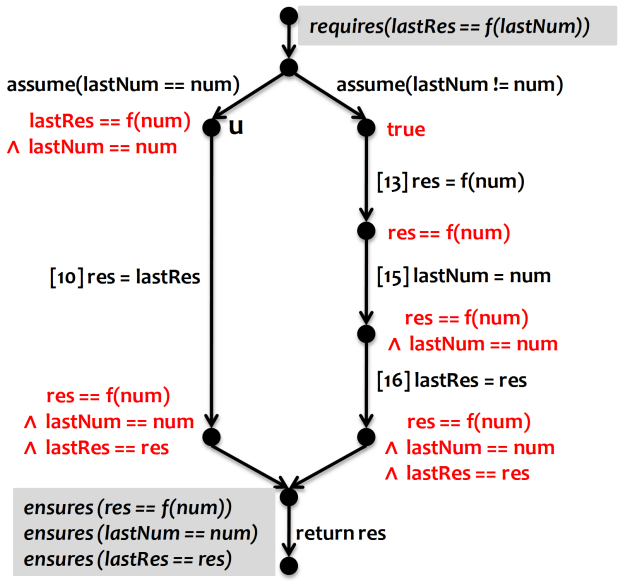

Figure 2: A control flow graph representation of the procedure `Compute`. Edges are labelled by program statements and nodes are labelled by formulas in the proof.

a piece of sequential code satisfies certain assertions in a sequential execution precisely identifies the properties relied on by the program at different points in execution; hence, such a sequential proof clearly identifies what concurrent interference can be permitted; thus, a correct concurrency control can be systematically (and even automatically) derived from such a proof.

We now provide an informal overview of our approach by illustrating how concurrency control can be synthesized for our running example. Figure 2 presents a proof of correctness for our running example (in a sequential setting). The program is presented as a control-flow graph, with its edges representing program statements. A proof consists of an invariant attached $\mu(u)$ to every vertex u in the control-flow graph, as illustrated in the figure such that: (a) for every edge $u \rightarrow v$ labelled with a statement s, execution of s in a state satisfying $\mu(u)$ is guaranteed to produce a state satisfying $\mu(v)$, and (b) for every edge $u \rightarrow v$ annotated with an assertion φ, we have $\mu(u) \Rightarrow \varphi$. Given a specification for a library's procedures, existing verification tools can be used to generate such proofs automatically. Though we discuss this aspect in the paper, the central focus of the paper is on using such a proof to synthesize concurrency control code.

Loosely speaking, the invariant $\mu(u)$ attached to a vertex u indicates the property required (by the proof) to hold at u in order to ensure that the procedure's execution satisfies all assertions of the procedure. We can reinterpret this in a concurrent setting as follows: when a thread t_1 is at point u, it can tolerate

changes to the state by another thread t_2 as long as the invariant $\mu(u)$ continues to hold from t_1's perspective; however, if another thread t_2 were to change the state such that t_1's invariant $\mu(u)$ is broken, then the continued execution by t_1 may fail to satisfy the desired assertions.

Consider the proof in Figure 2. The vertex labelled u in the figure corresponds to the point before the execution of statement [10][1] (after the if-condition evaluates to true). The invariant attached to this program point indicates that the proof of correctness depends on the condition[2] "`lastRes == f(num)`" holding true at this program point. The execution of statement [13] by another thread will not invalidate this condition. On the other hand, execution of statement [16] by another thread can potentially invalidate this condition. Thus, we infer that, when one thread is at point u, a concurrent execution of statement [16] (by another thread) should be avoided.

In general, assume that we want to avoid the execution of a statement s by one thread when another thread is at a program point u as s might invalidate a predicate p that is required at u. We can ensure this by introducing a lock $lock_p$ corresponding to p, and ensuring that every thread holds lock $lock_p$ at program point u and ensuring that every thread holds $lock_p$ when executing s. Note that the lock $lock_p$ does not have to correspond to any specific variable. It is a lock corresponding to a *predicate*.

Our algorithm for synthesizing concurrency control does precisely this. From the invariant $\mu(u)$ at vertex u, we compute a set of predicates $\mathtt{pm}(u)$. (For now, think of $\mu(u)$ as the conjunction of predicates in $\mathtt{pm}(u)$.) $\mathtt{pm}(u)$ represents the set of predicates required at u. For any edge $u \rightarrow v$, consider any predicate p that is in $\mathtt{pm}(v) \setminus \mathtt{pm}(u)$. This predicate is required at v but not at u. Hence, we acquire the lock for p along this edge. Dually, for any predicate that is required at u but not at v, we release the lock along the edge. Finally, if the execution of the statement in edge $u \rightarrow v$ can invalidate any predicate p (that is required at some point), we acquire and release the corresponding lock before and after the statement (unless it is already a required predicate at u or v).

Our algorithm ensures that the locking scheme does not introduce any deadlocks by merging locks where necessary, as we will describe later. Finally, we also optimize the produced solution using a few simple techniques. E.g., in our example whenever the lock m for `lastRes == res` is held, the lock l for `lastNum == num` is also held. Hence, we can eliminate the lock m as it is redundant.

Figure 1 shows the resulting library with the concurrency control we synthesize. It is easy to see that this implementation satisfies its specification even in a concurrrent setting. The concurreny control we infer permits a high degree to concurrency since it allows multiple threads to compute f concurrently. A more conservative but correct locking scheme would acquire the lock during the entire procedure.

One distinguishing aspect of our algorithm is that it involves very local rea-

[1] In the rest of this section, we shall use [l] as a shorthand for "at line numbered l".

[2] Our scheme permits treating the invariant as a single, complex, predicate such as (`lastRes == f(num)` $\wedge$ `lastNum == num`) or as a set of simpler predicates such as { `lastRes == f(num)`, `lastNum == num` }.

soning. In particular, it does not involve reasoning about interleaved executions, as is common with many analyses of concurrent programs.

Linearizability In general, the approach outlined above can be used to ensure thread-safety in any of the following senses: permit only interleavings that guarantee certain safety properties (such as the absence of null-pointer dereference) or preserve certain data-structure invariants, or even guarantee that procedures meet their specifications (e.g., given as a precondition/postcondition pair). While powerful, this may not be completely adequate. Ideally, the concurrent library implementation should enable the developer of a client of the library to reason about the client modularly, using the library's specification. For this reason, we believe that thread-safety should guarantee linearizability with respect to the sequential specification: any concurrent execution of a procedure should not only be guaranteed to satisfy its specification, it should also appear to take effect instantaneously at some point during its execution. In this paper, we show how the techniques sketched above can be extended to guarantee linearizability.

Logical interference Existing concurrency control mechanisms rely on a data-access based notion of interference: concurrent access to the same data, where at least one access is a write, is conservatively treated as undesirable interfence. This is true of pessimistic concurrency control mechanisms (such as those based on locking), which seek to avoid interference, as well as optimistic concurrency control mechanisms, which seek to detect interference after the fact and rollback. One of the contributions of this paper is that it introduces a more logical/semantic notion of interference and shows that it can be used to achieve more permissive, yet safe, concurrency control. Specifically, concurrency control based on this approach permits interleavings that existing schemes based on stricter notion of interference will disallow. Hand-crafted concurrent code often permits "benign interference" (e.g., racy accesses to the same data-item) for performance reasons. We believe that formalizing a logical notion of interference, as done in this paper, is useful for this reason.

While we present a lock-based pessimistic concurrency control mechanism, it would be interesting to explore the possibility of optimistic concurrency control mechanisms that exploit a similar weaker notion of interference.

2 The Problem

In this section, we formally define the problem setting, scope, and introduce required terminology. We introduce a simple language to illustrate our ideas, but our ideas are more broadly applicable.

2.1 The Sequential Setting

Sequential Libraries A library $\mathcal{L}$ is a pair $(\mathcal{P}, V_{\mathcal{L}})$, where $\mathcal{P}$ is a set of procedures of the form described below, and $V_{\mathcal{L}} = \{sv_1, \ldots, sv_n\}$ is a set of variables restricted to the scope of the library and shared by the procedures. We refer to variables in $V_{\mathcal{L}}$ as *global* variables. We only consider well-encapsulated libraries, i.e., none of the variables in $V_{\mathcal{L}}$ is visible to any procedure outside the library $\mathcal{L}$.

A procedure P is a pair (G_P, V_P), where G_P is a control-flow graph with each edge labelled by a primitive statement, and $V_P = \{lv_1, \ldots, lv_k\}$ is a set of variables restricted to the scope of the procedure. Every control-flow graph has a unique entry vertex N_P (which is assumed to have no predecessors) and a unique exit vertex X_P (which is assumed to have no successors). Primitive statements are either `skip` statements, assignment statements, `assume` statements, or `return` statements. An `assume` statement is used to implement conditional control flow in the usual way We refer to variables in V_P as *local* variables, and these include the formal parameters of the procedure as well as the procedure's local variables. To simplify the semantics, we will assume that the set V_P is the same for all procedures.

We use the notation $u \xrightarrow{s} v$ to identify an edge from u to v labelled with a primitive statement s.

We close an open library to obtain a whole program that captures all possible sequences of invocations of the library's procedures (by any and all clients) as follows. We define the *control graph* of a library to be the graph obtained by taking the union of the control-flow graphs of all the procedures, augmented by a new vertex w, as well as an edge from every procedure exit vertex to w and an edge from w to every procedure entry vertex. We refer to w as the *quiescent* vertex. Note that a one-to-one correspondence exists between a path in the control graph of the library, starting from w, and the execution of a sequence of procedure calls. In the semantics, an edge $w \rightarrow N_P$ from the quiescent vertex to the entry vertex of a procedure P is treated as representing all possible calls to procedure P. We will refer to these edges as *call edges*.

Sequential States We now present the operational semantics of our language. A procedure-local state σ_ℓ is a tuple $(pc, v_1, \ldots, v_n)$ where pc, the program counter, is a vertex in the library graph and each v_i represents the value of local variable lv_i. We represent the set of all procedure-local states by Σ^s_ℓ. A global state σ_g is a tuple $(v_1, \ldots, v_k)$, where each v_i is the value of global variable sv_i. Let Σ^s_g represent the set of all global states. A library state σ is a pair (σ_ℓ, σ_g), where σ_ℓ is a procedure-local state, and σ_g is a global state that is persistent across procedure calls. The set of all library states is thus $\Sigma^s = \Sigma^s_\ell \times \Sigma^s_g$. We say that a state is a *quiescent state* if its pc value is w and that it is a *entry state* if its pc value equals the entry vertex of some procedure.

Sequential Executions The semantics of the library can be captured as a transition relation $\rightsquigarrow_s \subseteq \Sigma^s \times \Sigma^s$ as follows. Every control-flow edge e induces

a transition relation $\overset{e}{\rightsquigarrow}_s$, where $\sigma \overset{e}{\rightsquigarrow}_s \sigma'$ iff the execution of (the statement labelling) edge e transforms state σ to σ'. (In particular, note that this implies that the *pc* of σ must be the source of e and that the *pc* of σ' must be the target of e.) The edge $w \rightarrow N_P$ from the quiescent vertex to the entry vertex of a procedure P models an arbitrary call to procedure P. Hence, in defining the transition relation, such edges are treated as statements that assign a non-deterministically chosen value to every formal parameter of P and the default initial value to every local variable of P. Similarly, the edge $X_P \rightarrow w$ is treated as a `skip` statement. We say $\sigma \rightsquigarrow_s \sigma'$ if there exists some edge e such that $\sigma \overset{e}{\rightsquigarrow}_s \sigma'$.

We define a *sequential execution* to be a sequence of states $\sigma_0\sigma_1 \cdots \sigma_k$ where σ_0 is the initial state of the library and we have $\sigma_i \rightsquigarrow_s \sigma_{i+1}$ for $0 \leq i < k$. A sequential execution represents the execution of a sequence of procedure calls, one after another, by the library (where the last call's execution may be incomplete). Given a sequential execution $\sigma_0\sigma_1 \cdots \sigma_k$, we say that σ_i is the *corresponding entry state* of σ_j if σ_i is an entry state and no state σ_h is an entry state for $i < h \leq j$.

A statement of the form $u \overset{\texttt{assume cond}}{\rightarrow} v$ has the semantics that execution of P proceeds to the subsequent statement only for those states that satisfy the condition `cond`.

Sequential Assertions and Specifications We augment the underlying language with an `assert` statement, a new kind of primitive statement that specifies a state-level assertion using predicates over program states. We use the notation $\sigma \models_s \varphi$ to denote that a state σ satisfies the assertion φ. Control-flow edges may now be annotated with such assertions as well. Note that `assert` statements have no effect on the execution semantics: we extend the sequential execution semantics to treat `assert` statements as `skip` statements. Assertions are used only to define the notion of *well-behaved* executions as follows.

Definition 1. *A sequential execution π satisfies the library's assertions if for any transition $\sigma_i \overset{\texttt{assert } \varphi}{\rightsquigarrow_s} \sigma_{i+1}$ in the execution, we have $\sigma_i \models_s \varphi$. A sequential library satisfies its assertions if every execution of the library satisfies its assertions.*

Single-state assertions of the above form can be used to specify the invariants expected at certain program points. They can even be used to provide functional specifications of procedures in some cases (as in our running example). However, in general, functional specifications take the form of two-state assertions, which relate the input state to output state. So, we enrich the language of assertions to permit specification of such two-state assertions, by allowing assertions to use special *input variables* v^{in} (to refer to the value of the variable in the first state). E.g., the specification "$x == x^{in} + 1$" asserts that the value of x in the second state is one more than its value in the first state. The semantics of such a specification Φ is defined via a relation $(\sigma_{in}, \sigma_{out}) \models_s \Phi$.

Definition 2. *A sequential execution π satisfies the library's specifications if for any transition $\sigma_i \overset{\texttt{assert}\ \Phi}{\rightsquigarrow_s} \sigma_{i+1}$ in the execution, we have $(\sigma_{in}, \sigma_i) \models_s \Phi$ where σ_{in} is the corresponding entry state of σ_i. A sequential library satisfies its specifications if every execution of the library satisfies its specifications.*

2.2 The Concurrent Setting

Concurrent Libraries Recall that our goal is to augment a sequential library with concurrency control to ensure that it continues to perform correctly. This motivates the concurrent language we present now, which augments the sequential language with lock-based concurrency control constructs.

A concurrent library $\mathcal{L}$ is a triple $(\mathcal{P}, V_{\mathcal{L}}, Lk)$, where $\mathcal{P}$ is a set of concurrent procedures, $V_{\mathcal{L}}$ is a set of global variables, and Lk is a set of locks. A concurrent procedure is like a sequential procedure, with the extension that a primitive statement is either a sequential primitive statement or a locking statement of the form `acquire(`ℓ`)` or `release(`ℓ`)` where ℓ is a lock.

Concurrent States A concurrent library permits concurrent (i.e., overlapping) invocations of procedures. We associate each procedure invocation with a thread (representing the client thread that invoked the procedure). Let T denote an infinite set of thread-ids, which are used as unique identifiers for threads. (These may be thought of as representing an unbounded pool of threads used to process procedure call invocations.)

In a concurrent execution, every thread has a private copy of local variables, but all threads share a single copy of the global variables. Hence, the local-state in a concurrent execution is represented by a map from T to Σ^s_ℓ. (A thread whose local-state's *pc* value is the quiescent point represents an idle thread, i.e., a thread not processing any procedure invocation.) We denote the set of all local states by Σ^c_ℓ. Thus, $\Sigma^c_\ell = T \to \Sigma^s_\ell$.

All locks are effectively global variables. At any point during execution, a lock lk is either free or held by one particular thread. We represent the state of locks during execution by a partial function from Lk to T indicating which thread, if any, holds any given lock. Let $\Sigma^c_{\text{lk}} = Lk \hookrightarrow T$ represent the set of all lock-states. We denote the set of all global states by Σ^c_g, and define Σ^c_g to be $\Sigma^s_g \times \Sigma^c_{\text{lk}}$. Thus, the set of all states of $\mathcal{L}$ in the concurrent setting is $\Sigma^c = \Sigma^c_\ell \times \Sigma^c_g$. Given a concurrent state $\sigma = (\sigma_\ell, (\sigma_g, \sigma_{lk}))$ and thread t, we define $\sigma[t]$ to be the sequential state $(\sigma_\ell(t), \sigma_g)$.

Concurrent Executions We can capture the concurrent execution semantics as transition relations as well. Let e be any control-flow edge labelled with a sequential primitive statement, and t be any thread. We say that $(\sigma_\ell, (\sigma_g, \sigma_{lk})) \overset{(t,e)}{\rightsquigarrow}_c (\sigma'_\ell, (\sigma'_g, \sigma_{lk}))$ iff $(\sigma_t, \sigma_g) \overset{e}{\rightsquigarrow}_s (\sigma'_t, \sigma'_g)$ where $\sigma_t = \sigma_\ell(t)$ and $\sigma'_\ell = \sigma_\ell[t \mapsto \sigma'_t]$. The transitions corresponding to lock acquire/release are defined in the obvious way. We say that $\sigma \rightsquigarrow_c \sigma'$ iff there exists some (t, e) such that $\sigma \overset{(t,e)}{\rightsquigarrow}_c \sigma'$.

We define a *concurrent execution* to be a sequence $\sigma_0\sigma_1\cdots\sigma_k$, where σ_0 is the initial state of the library and $\sigma_i \overset{\ell_i}{\rightsquigarrow}_c \sigma_{i+1}$ for every $0 \leq i < k$. We say that $\ell_0\cdots\ell_{k-1}$ is the *schedule* of this execution. Furthermore, we say that any sequence $\ell\cdots\ell_m$ is a *feasible schedule* if it is the schedule of some concurrent execution. Consider a concurrent execution $\sigma_0\sigma_1\cdots\sigma_k$. We say that a state σ_i is a t-entry-state if it is generated from a quiescent state by thread t executing a call edge. We say that σ_i is the *corresponding t-entry state* of σ_j if σ_i is a t-entry-state and no state σ_h is a t-entry-state for $i < h \leq j$.

Interpreting Assertions and Specifications In Concurrent Executions Recall that we denote satisfaction of an assertion φ by a sequential state σ by $\sigma \models_s \varphi$. In a concurrent setting, assertions are evaluated in the context of the thread that executes the corresponding `assert` statement. We say that state σ satisfies an assertion φ in the context of thread t_i (denoted by $(\sigma, t_i) \models_c \varphi$) iff $\sigma[t_i] \models_s \varphi$.

We extend the above definition to two-state assertions in a similar fashion. For any specification Φ, we say that a given pair of states $(\sigma_{in}, \sigma_{out})$ satisfies Φ in the context of thread t (denoted by $((\sigma_{in}, \sigma_{out}), t) \models_c \Phi$) iff $(\sigma_{in}[t], \sigma_{out}[t]) \models_s \Phi$.

Definition 3. *A concurrent execution π satisfies the library's specifications if for any transition $\sigma_i \overset{(t,\texttt{assert}\ \Phi)}{\rightsquigarrow_c} \sigma_{i+1}$ in the execution, we have $((\sigma_{in}, \sigma_i), t) \models_c \Phi$ where σ_{in} is the corresponding t-entry state of σ_i. A concurrent library satisfies its specifications if every execution of the library satisfies its specifications.*

Frame Conditions Consider a library with two global variables `x` and `y`. Consider a procedure `IncX` whose specification is that it will increment the value of `x` by 1. One possible formalization of this specification is $(x == x^{in} + 1)$ && $(y == y^{in})$. The condition $y == y^{in}$ is `IncX`'s frame condition, which says that it will not modify `y`. Our above formalization of concurrent specification correctness becomes unnecessarily restrictive if frame conditions are explicitly stated as above. (A concurrent update to `y` by another procedure, when `IncX` is executing, would be considered a violation of `IncX`'s specification.) Our formalization can be generalized to handle frame conditions better by treating a specification as a pair (S, Φ) where S is the set of all global variables referenced (read or updated) by any execution of the procedure, and Φ is a specification that does not refer to any global variables outside S. For our above example, the specification will be ($\{\texttt{x}\}$, *ensures* $x == x^{in} + 1$)).

2.3 Goals

Our goal, formally, is: Given a sequential library $\mathcal{L}$ annotated with assertions satisfied by $\mathcal{L}$ in every sequential execution, construct $\hat{\mathcal{L}}$, by augmenting $\mathcal{L}$ with concurrency control, such that every concurrent execution of $\hat{\mathcal{L}}$ satisfies all assertions, both single-state and two state assertions. In Section 5, we extend this goal to construct $\hat{\mathcal{L}}$ such that every concurrent execution of $\hat{\mathcal{L}}$ is linearizable.

3 Preserving Single-State Assertions

In this section we describe our algorithm for synthesizing concurrency control, but restrict our attention to single-state assertions.

3.1 Algorithm Overview

Definitions. A *predicate mapping* is a mapping $\mathtt{pm}$ from the vertices of G_P to a set of predicates involving variables in $V_{\mathcal{L}} \cup V_P$. A *sequential proof* is a mapping μ from vertices of G_P to formulae such that (a) for every edge $u \xrightarrow{s} v$, $\{\mu(u)\}s\{\mu(v)\}$ is a valid Hoare triple, and (b) for every edge $u \xrightarrow{\texttt{assert } \varphi} v$, we have $\mu(u) \Rightarrow \varphi$.

Note that the invariant $\mu(u)$ attached to a vertex u by a proof indicates two things: (i) any sequential execution reaching point u will produce a state satisfying $\mu(u)$, and (ii) any sequential execution from point u, starting from a state satisfying $\mu(u)$ will satisfy the invariants labelling other program points (and satisfy all assertions encountered during the execution).

A procedure that satisfies its assertions in a sequential execution may fail to do so in a concurrent execution due to interference by other threads. E.g., consider a thread t_1 that reaches a program point u in a state that satisfies $\mu(u)$. At this point, another thread t_2 may execute some statement that changes the state to one where $\mu(u)$ no longer holds. Now, we no longer have a guarantee that a continued execution by t_1 will successfully satisfy its assertions.

The above reasoning also hints at a solution to this problem: when a thread t_1 is at point u, we should ensure that no other thread t_2 changes the state to one where t_1's invariant $\mu(u)$ fails to hold. Any change to the state by another thread t_2 can be tolerated by t_1 *as long as the invariant* $\mu(u)$ *continues to hold*. We can achieve this by associating a lock with the invariant $\mu(u)$, ensuring that t_1 holds this lock when it is at program point u, and ensuring that any thread t_2 acquires this lock before executing a statement that may cause this invariant to be broken. An invariant $\mu(u)$, in general, may be a boolean formula over simpler predicates. We could potentially get different locking solutions by associating different locks with different sub-formulae of the invariant. We now introduce some definitions to present our solution formally in a generalized setting.

A predicate mapping $\mathtt{pm}$ is a *basis* for a proof μ if every $\mu(u)$ can be expressed as a boolean formula (involving conjunctions, disjunctions, and negation) over $\mathtt{pm}(u)$. A basis $\mathtt{pm}$ for proof μ is *positive* if every $\mu(u)$ can be expressed as a boolean formula involving only conjunctions and disjunctions over $\mathtt{pm}(u)$. Consider a sequential execution that reaches program point u in state σ. Not all predicates in $\mathtt{pm}(u)$ may hold true in state σ. However, our earlier intuition applies to basis predicates as well: we can still tolerate any concurrent update to the state as long as no basis predicate is falsified (changed from true to false).

Given a proof μ, we say that an edge $u \xrightarrow{s} v$ *sequentially positively preserves* a predicate φ if $\{\mu(u) \wedge \varphi\}s\{\varphi\}$ is a valid Hoare triple. Otherwise, we say that the edge *may sequentially falsify* the predicate φ. Note that the above definition is in terms of the Hoare logic for our sequential language (semantics), which

explains our use of the adjective "sequentially". What we want to formalize is the notion of a thread t_2's execution of an edge falsifying a predicate φ in a thread t_1's scope. Given a predicate φ, let $\hat{\varphi}$ denote the predicate obtained by replacing every local variable x with a new unique variable $\hat{x}$. We say that an edge $u \xrightarrow{s} v$ *may falsify* φ iff the edge may sequentially falsify $\hat{\varphi}$. (Note that this reasoning requires working with formulas with free variables, such as $\hat{x}$. This is, however, straight-forward as the free variables can be handled just like extra program variables.)

E.g., consider statement `lastRes = res` at line [16] in Fig. 1. Consider predicate `lastRes == f(num)`. By renaming local variable `num` to avoid naming conflicts, we obtain predicate `lastRes == f(`$n\hat{u}m$`)`. We say that the statement at line [16] *may falsify* this predicate because the triple $\{res == f(num) \wedge lastNum == num \wedge lastRes == f(n\hat{u}m)\}$ `lastRes = res` $\{lastRes == f(n\hat{u}m)\}$ is not a valid Hoare triple.

Let $\mathfrak{pm}$ be a positive basis for a proof μ and $\mathcal{R} = \cup_u \mathfrak{pm}(u)$. If a predicate φ is in $\mathfrak{pm}(u)$, we say that φ is *relevant* at program point u. In a concurrent execution, we say that a predicate φ is relevant to a thread t in a given state if t is at a program point u in the given state and $\varphi \in \mathfrak{pm}(u)$. Our locking scheme associates a lock with every predicate φ in $\mathcal{R}$. The invariant it establishes is that a thread, in any state, will hold the locks corresponding to precisely the predicates that are relevant to it. We will simplify the initial description of our algorithm by assuming that distinct predicates are associated with distinct locks and later relax this requirement.

Consider any control-flow edge $e = u \xrightarrow{s} v$. Consider any predicate φ in $\mathfrak{pm}(v) \setminus \mathfrak{pm}(u)$. We say that predicate φ *becomes relevant* at edge e. In the motivating example, the predicate `lastNum == num` becomes relevant at the statement at line [15].

We ensure the desired invariant by adding an acquire of the locks corresponding to every predicate that becomes relevant at edge e. This acquire is added prior to statement s in the edge. (Acquiring the lock after s may be too late, as some other thread could intervene between s and the acquire and break predicate φ.)

Now consider any predicate φ in $\mathfrak{pm}(u) \setminus \mathfrak{pm}(v)$. We say that φ *becomes irrelevant* at edge e. E.g., predicate `lastres == f(lastNum)` becomes irrelevant once the false branch at line [9] is taken. For every p that becomes irrelevant at edge e, we add a release of the lock corresponding to p *after* statement s.

The above steps ensure that in a concurrent execution a thread will hold a lock on all predicates relevant to it. The second component of the concurrency control mechanism is to ensure that any thread that acquires a lock on any predicate before it falsifies the predicate. Consider an edge $e = u \xrightarrow{s} v$ in the control-flow graph. Consider any predicate $\varphi \in \mathcal{R}$ that may be falsified by edge e. We add an acquire of the lock corrresponding to this predicate before s (unless $\varphi \in \mathfrak{pm}(u)$), and add a release of the same lock after s (unless $\varphi \in \mathfrak{pm}(v)$).

Managing locks at procedure entry/exit We will need to acquire/release locks at procedure entry and exit differently from the scheme above. Our algorithm works with the control graph defined in Section 2. Recall that we use a quiescent vertex w in the control graph. The invariant $\mu(w)$ attached to this quiescent vertex describes invariants maintained by the library (in between procedure calls). Any `return` edge $u \overset{return}{\rightarrow} v$ must be augmented to release all locks corresponding to predicates in $\mathfrak{pm}(u)$ before returning. Dually, any procedure entry edge $w \rightarrow u$ must be augmented to acquire all locks corresponding to predicates in $\mathfrak{pm}(u)$.

However, this is not enough. Let $w \rightarrow u$ be a procedure p's entry edge. The invariant $\mu(u)$ is part of the library invariant that procedure p depends upon. It is important to ensure that when p executes the entry edge (and acquires locks corresponding to the basis of $\mu(u)$) the invariant $\mu(u)$ holds. We achieve this by ensuring that any procedure that invalidates the invariant $\mu(u)$ holds the locks on the corresponding basis predicates until it reestablishes $\mu(u)$. We now describe how this can be done in a simplified setting where the invariant $\mu(u)$ can be expressed as the conjunction of the predicates in the basis $\mathfrak{pm}(u)$ for every procedure entry vertex u. (Disjunction can be handled at the cost of extra notational complexity.) We will refer to the predicates that occur in the basis $\mathfrak{pm}(u)$ of some procedure entry vertex u as *library invariant predicates*.

We use an *obligation* mapping $\mathfrak{om}(v)$ that maps each vertex v to a set of library invariant predicates to track the invariant predicates that may be invalid at v and need to be reestablished before the procedure exit. We say a function $\mathfrak{om}$ is a valid obligation mapping if it satisfies the following constraints for any edge $e = u \rightarrow v$: (a) if e may falsify a library invariant φ, then φ must be in $\mathfrak{om}(v)$, and (b) if $\varphi \in \mathfrak{om}(v)$, then φ must be in $\mathfrak{om}(v)$ unless e *establishes* φ. Here, we say that an edge $u \overset{s}{\rightarrow} v$ *establishes* a predicate φ if $\{\mu(u)\}s\{\varphi\}$ is a valid Hoare triple. Define $\mathfrak{m}(u)$ to be $\mathfrak{pm}(u) \cup \mathfrak{om}(u)$. Now, the scheme described earlier can be used, except that we use $\mathfrak{m}$ in place of $\mathfrak{pm}$.

Locking along `assume` edges Recall that we model conditional branching, based on a condition p, using two edges labelled "`assume` p" and "`assume` $!p$". Any lock to be acquired along an `assume` edge will need to be acquired before the condition is evaluated. If the lock is required along both `assume` edges, this is sufficient. If it is required along only one `assume` edge, then we will have to release the lock along the edge where it is not required.

Deadlock Prevention The locking scheme synthesized above may potentially lead to a deadlock. We now show how to modify the locking scheme to avoid this possibility. For any edge e, let $\mathfrak{mbf}(e)$ be (a conservative approximation of) the set of all predicates that may be falsified by the execution of edge e. We first define a binary relation $\rightarrowtail$ on the predicates used (i.e., the set $\mathcal{R}$) as follows: we say that $p \rightarrowtail r$ iff there exists a control-flow edge $u \overset{s}{\rightarrow} v$ such that $p \in \mathfrak{m}(u) \wedge r \in (\mathfrak{m}(v) \cup \mathfrak{mbf}(u \overset{s}{\rightarrow} v)) \setminus \mathfrak{m}(u)$. Note that $p \rightarrowtail r$ holds iff it is possible for some thread to try to acquire a lock on r while it holds a lock on p.

Let $\rightarrowtail^*$ denote the transitive closure of $\rightarrowtail$.

We define an equivalence relation $\rightleftarrows$ on $\mathcal{R}$ as follows: $p \rightleftarrows r$ iff $p \rightarrowtail^* r \wedge r \rightarrowtail^* p$. Note that any possible deadlock must involve an equivalence class of this relation. If we map all predicates in an equivalence class to the same lock, we can avoid deadlocks. In addition to the above, we establish a total ordering on all the locks, and ensure that all lock acquisitions we add to a single edge are done in an order consistent with the established ordering.

Optimizations We now discuss some optimizations that can improve the basic locking scheme and result in better performance.

Our scheme can sometimes introduce *redundant* locking. E.g., assume that in the generated solution a lock ℓ_1 is always held whenever a lock ℓ_2 is acquired. Then, the lock ℓ_2 is redundant and can be eliminated. Similarly, if we have a predicate φ that is never falsified by any statement in the library, then we do not need to acquire a lock for this predicate. We can eliminate such redundant locks as a final optimization pass over the generated solution.

Note that it is safe for multiple threads to simultaneously hold a lock on the same predicate φ if they want to "preserve" it, but a thread that wants to "break" φ needs an exclusive lock. Thus, we can use reader-writer locks to improve concurrency. However, since it is unsafe for a thread that holds a read-lock on a predicate φ to try to acquire a write-lock φ, using this optimization also requires an extension to the basic deadlock avoidance scheme.

Proofs via Predicate Abstraction The sequential proof required by our scheme can be generated using verification tools, e.g., predicate-abstraction based tools such as SLAM [3], BLAST [10] and Synergy [9]. Since a minimal proof and a minimal basis, in general, can lead to better concurrency control, approaches such as BLAST and Synergy are preferable since they construct lazy and local abstractions, where a predicate is added to a set $\mathfrak{pm}(u)$ as and when necessary.

3.2 Complete Schema

We now present a complete outline of our schema for synthesizing concurrency control.

1. Construct a sequential proof μ that the library satisfies the given assertions in any sequential execution.
2. Construct a positive basis $\mathfrak{pm}$ and an obligation mapping $\mathfrak{om}$ for the proof μ.
3. Compute a map $\mathfrak{mbf}$ from the edges of the control graph to $\mathcal{R}$, the range of $\mathfrak{pm}$, such that $\mathfrak{mbf}(e)$ (conservatively) includes all predicates in $\mathcal{R}$ that may be falsified by the execution of e.
4. Compute the equivalence relation $\rightleftarrows$ on $\mathcal{R}$.

5. Generate a predicate lock allocation map $\mathfrak{lm} : \mathcal{R} \to \mathcal{L}$ such that for any $\varphi_1 \rightleftarrows \varphi_2$, we have $\mathfrak{lm}(\varphi_1) = \mathfrak{lm}(\varphi_2)$.

6. Compute the following quantities for every edge $e = u \xrightarrow{s} v$, where we use $\mathfrak{lm}(X)$ as shorthand for $\{\ \mathfrak{lm}(p) \mid p \in X\ \}$ and $\mathfrak{m}(u) = \mathfrak{pm}(u) \cup \mathfrak{om}(u)$:

$$\begin{array}{lcl} \mathit{BasisLocksAcq}(e) & = & \mathfrak{lm}(\mathfrak{m}(v)) \setminus \mathfrak{lm}(\mathfrak{m}(u)) \\ \mathit{BasisLocksRel}(e) & = & \mathfrak{lm}(\mathfrak{m}(u)) \setminus \mathfrak{lm}(\mathfrak{m}(v)) \\ \mathit{BreakLocks}(e) & = & \mathfrak{lm}(\mathfrak{mbf}(e)) \setminus \mathfrak{lm}(\mathfrak{m}(u)) \setminus \mathfrak{lm}(\mathfrak{m}(v)) \end{array}$$

7. We obtain the concurrency-safe library $\widehat{\mathcal{L}}$ by transforming every edge $u \xrightarrow{s} v$ in the library $\mathcal{L}$ as follows:

   (a) for every predicate p in $\mathit{BasisLocksAcq}(u \xrightarrow{s} v)$, we add an `acquire(`$\mathfrak{lm}(p)$`)` before s;

   (b) for every predicate p in $\mathit{BasisLocksRel}(u \xrightarrow{s} v)$, we add a `release(`$\mathfrak{lm}(p)$`)` after s;

   (c) for every predicate p in $\mathit{BreakLocks}(u \xrightarrow{s} v)$, we add an `acquire(`$\mathfrak{lm}(p)$`)` before s and a `release(`$\mathfrak{lm}(p)$`)` after s.

   All lock acquisitions along a given edge are added in an order consistent with a total order established on all locks.

3.3 Correctness

We now present a formal statement of the correctness claims for our algorithm. Let $\mathcal{L}$ be a given library with a set of embedded assertions satisfied by all sequential executions of $\mathcal{L}$. Let $\widehat{\mathcal{L}}$ be the library obtained by augmenting $\mathcal{L}$ with concurrency control using the algorithm presented in Section 3.2. Let μ, $\mathfrak{pm}$, and $\mathfrak{om}$ be the proof, the positive basis and the obligation map used to generate $\widehat{\mathcal{L}}$.

Consider any concurrent execution of the given library $\mathcal{L}$. We say that a thread t is *safe* in a state σ if $(\sigma, t) \models_c \mu(u)$ where t's program-counter in state σ is u. We say that thread t is *active* in state σ if it's program-counter is something other than the quiescent vertex. We say that state σ is *safe* if every active thread t in σ is safe. Recall that a concurrent execution is of the form: $\sigma_0 \xrightarrow{\ell_0} \sigma_1 \xrightarrow{\ell_1} \cdots \sigma_n$, where each label ℓ_i is an ordered pair (t, e) indicating that the transition is generated by the execution of edge e by thread t. We say that a concurrent execution is safe if every state in the execution is safe. It trivially follows that a safe execution satisfies all assertions of $\mathcal{L}$.

Note that every concurrent execution π of $\widehat{\mathcal{L}}$ corresponds to an execution π' of $\mathcal{L}$ if we ignore the transitions corresponding to lock acquire/release instructions. We say that an execution π of $\widehat{\mathcal{L}}$ is safe if the corresponding execution π' of $\mathcal{L}$ is safe. The goal of the concurrency control is to ensure that all possible executions of $\widehat{\mathcal{L}}$ are safe.

We say that a transition $\sigma \xrightarrow{(t,e)} \sigma'$ is *interference-free* if for every active thread $t' \neq t$ whose program-counter in state σ is u (a) for every predicate $\varphi \in \mathtt{pm}(u)$ the following holds: if $(\sigma, t') \models_c \varphi$, then $(\sigma', t') \models_c \varphi$, and (b) if $e = x \rightarrow y$ is an entry edge, then none of the predicates in $\mathtt{pm}(y)$ are in $\mathtt{om}(u)$. A concurrent execution is termed interference-free if all transitions in the execution are interference-free.

Theorem 1. *(a) Any interference-free concurrent execution of $\mathcal{L}$ is safe. (b) Any concurrent execution of $\widehat{\mathcal{L}}$ corresponds to an interference-free execution of $\mathcal{L}$. (c) Any concurrent execution of $\widehat{\mathcal{L}}$ satisfies every assertion of $\mathcal{L}$. (d) The library $\widehat{\mathcal{L}}$ is deadlock-free.*

Proof. (a) We prove that every state in an interference-free execution of $\mathcal{L}$ is safe by induction on the length of the execution.

Consider a thread t in state σ with program-counter value u. Assume that t is safe in σ. Thus, $(\sigma, t) \models_c \mu(u)$. Note that $\mu(u)$ can be expressed in terms of the predicates in $\mathtt{pm}(u)$ using conjunction and disjunction. Let SP denote the set of all predicates φ in $\mathtt{pm}(u)$ such that $(\sigma, t) \models_c \varphi$. Let σ' be any state such that $(\sigma', t) \models_c \varphi$ for every $\varphi \in SP$. Then, it follows that t is safe in σ'. Thus, it follows that after any interference-free transition every thread that was safe before the transition continues to be safe after the transition.

We now just need to verify that whenever an inactive thread becomes active (representing a new procedure invocation), it starts off being safe. We can establish this by inductively showing that every library invariant must be satisfied in a given state or must be in $\mathtt{om}(u)$ for some active thread t at vertex u.

(b) Consider a concurrent execution of $\widehat{\mathcal{L}}$. We need to show that every transition in this execution, ignoring lock acquires/releases, is interference-free. This follows directly from our locking scheme. Consider a transition $\sigma \xrightarrow{(t,e)} \sigma'$. Let $t' \neq t$ be an active thread whose program-counter in state σ is u. For every predicate $\varphi \in \mathtt{pm}(u) \cup \mathtt{om}(u)$, our scheme ensures that t' holds the lock corresponding to φ. As a result, neither of the conditions for interference can be satisfied.

(c) This follows immediately from (a) and (b).

(d) This follows from our scheme for merging locks. □

4 Extensions For 2-State Assertions

We now show how the algorithm presented in the previous section can be extended to ensure that any concurrent execution of the library will satisfy given 2-state assertions. As explained earlier, 2-state assertions are useful in writing functionality specifications.

Instrumented semantics. We now define an instrumented semantics that will allow us to treat these 2-state assertions (in the original semantics) as single-state assertions (in the instrumented semantics). Informally, the instrumented

semantics corresponds to the following program transformation. We augment the set of local variables with a new variable $\tilde{v}$ for every (local or shared) variable v in the original program and add a primitive statement $\mathcal{LP}$ at the entry of every procedure, whose execution essentially copies the value of every variable v to the corresponding instrumented variable $\tilde{v}$.

The Hoare logic for the standard semantics can be extended to this instrumented semantics in a straightforward fashion. The semantics of the statement $\mathcal{LP}$, in Hoare logic, is specified by: $\{\varphi[\tilde{x} \mapsto x, \cdots]\}\ \mathcal{LP}\ \{\varphi\}$. In other words, the weakest-precondition of φ, with respect to $\mathcal{LP}$, is obtained by replacing every instrumentation variable in φ by the corresponding base variable.

Let $\underline{\sigma'}$ denote the projection of an instrumented-semantics state σ' to a state in the standard semantics obtained by forgetting the values of the instrumentation variables. Given a 2-state assertion Φ, let $\tilde{\Phi}$ denote the single-state assertion in the instrumented semantics obtained by replacing every v^{in} by $\tilde{v}$. As formalized by the claim below, the satisfaction of a 2-state assertion Φ by executions in the standard semantics corresponds to satisfaction of the single-state assertion $\tilde{\Phi}$ in the instrumented semantics.

Lemma 1. *(a) A schedule ξ is feasible in the instrumented semantics iff it is feasible in the standard semantics. (b) Let σ' and σ be the states produced by a particular schedule with the instrumented and standard semantics, respectively. Then, $\sigma = \underline{\sigma'}$. (c) Let π' and π be the executions produced by a particular schedule with the instrumented and standard semantics, respectively. Then, π satisfies a single-state assertion φ iff π' satisfies it. Furthermore, π satisfies a 2-state assertion Φ iff π' satisfies the corresponding one-state assertion $\tilde{\Phi}$.*

In the sequel, we will not distinguish between v^{in} and the instrumentation variable $\tilde{v}$ and use just v^{in}.

Synthesizing concurrency control. Since 2-state assertions reduce to single state assertions in the instrumented semantics, we can now apply the techniques discussed in Section 3 for synthesizing concurrency control that preserves single-state assertions. Note that, in this setting, predicates may also involve variables of the form x^{in}. However, this just represents the local variable $\tilde{v}$ and the notions of a proof, basis, and and breaking predicates are the same as before.

5 Guaranteeing Linearizability

In the previous section, we showed how we can derive concurrency control to ensure that each procedure satisfies its sequential specification even in a concurrent execution. However, this may still be too permissive, allowing interleaved executions that produce counter-intuitive results. E.g., consider the procedure `Increment` shown in Figure 3, which increments a shared variable `x` by `1`. The figure shows the concurrency control derived using our approach to ensure specification correctness. Now consider a multi-threaded client that initializes `x` to `0` and invokes `Increment` concurrently in two threads. It would be natural to

```
1 global int x;
2 //@ensures x == x^in + 1 ∧ returns x
3 Increment() {
4   int tmp;
5   acquire(l_(x==x^in)); tmp = x; release(l_(x==x^in));
6   tmp = tmp + 1;
7   acquire(l_(x==x^in)); x = tmp; release(l_(x==x^in));
8   return tmp;
9 }
```

Figure 3: A non-linearizable implementation of the procedure `Increment`

expect that the value of `x` would be `2` at the end of any execution of this client. However, this implementation permits an interleaving in which the value of `x` at the end of the execution is `1`: the problem is that both invocations of `Increment` individually meet their specifications, but the cumulative effect is unexpected.

This is one of the difficulties with using pre/post-condition specifications to reason about concurrent executions. We can enable clients to reason about concurrent executions in a modular fashion by guaranteeing that the concurrent library is *linearizable* [11]. with respect to its sequential specification. In this section, we formally define linearizability and we show how our approach can be extended to derive concurrency control mechanisms that guarantee linearizability.

5.1 Linearizability

We now adapt and extend the earlier notation to define the notion of linearizability. Without loss of generality, we assume that each procedure returns the value of a special local variable *ret*.

Linearizability is a property of the library's externally observed behavior. A library's interaction with its clients can be described in terms of a *history*, which is a sequence of events, where each event is an *invocation* event or a *response* event. An invocation event is a tuple consisting of the procedure invoked, the input parameter values for the invocation, as well as a unique id. A response event consists of the id of a preceding invocation event, as well as a return value. Furthermore, an invocation event can have at most one matching response event. A complete history has a matching response event for every invocation event.

Consider sequential executions. A sequential history consists of an alternating sequence $inv_1, r_1, \cdots, inv_n, r_n$ of invocation events and corresponding response events. We will abuse our earlier notation and use $\sigma + inv_i$ to denote an entry state corresponding to a procedure invocation consisting of a valuation σ for the library's global variables and a valuation inv_i for the invoked procedure's formal parameters. We will similarly use $\sigma + r_i$ to denote a procedure exit state with return value r_i. Let σ_0 denote the value of the globals in the library's initial state. Let Φ_i denote the specification of the procedure invoked by inv_i. We say that a sequential history is *legal* if there exist valuations σ_i,

$1 \leq i \leq n$, for the library's globals such that $(\sigma_{i-1} + inv_i, \sigma_i + r_i) \models_s \Phi_i$ for $1 \leq i \leq n$.

We say that a complete interleaved history H is *linearizable* if there exists some legal sequential history S such that (a) H and S have the same set of invocation and response events and (b) for every return event r that precedes an invocation event *inv* in H, r and *inv* appear in that order in S as well. An incomplete history H is said to be linearizable if the complete history H' obtained by appending some response events and omitting some invocation events without a matching response event is linearizable.

Finally, a library $\mathcal{L}$ is said to be linearizable if every history produced by $\mathcal{L}$ is linearizable.

5.2 Linearization Points

In the sequel, we present an algorithm for synthesizing concurrency control that guarantees linearizability, which takes a linearization point specification (defined below) as an extra parameter, inspired by the classical notion of *linearization points*. We define a *linearization point set*, for a procedure, to be a set S of edges in its control-flow graph such that (a) every path from the entry vertex to exit vertex passes through some edge in S and (b) the procedure does not update any global variable along any path from the entry vertex to any edge in S. A *linearizability point specification* consists of a linearization point set for every procedure in the library.

In the sequel, the reader may assume the simple linearizability point specification consisting of the entry edge for every procedure. (Handling a more general linearizability point specification requires treating a reference to x^{in} in a postcondition as denoting the value of x when the procedure invocation executes its linearization point. Correspondingly, in our algorithm, we adapt the control-flow graph representation by instrumenting every linearization point (edge) by the statement $\mathcal{LP}$ defined in Section 4.)

5.3 Synthesizing concurrency control for linearizability

We now show how our approach can be extended to guarantee linearizability modulo a sequential specification and a linearization point specification. Our ambitious approach of locking predicates, rather than data-items, makes this goal challenging, as ideas from conventional approaches (based on locking data-items) do not necessarily carry over. Thus, we study few tricky cases and shed light on the problem of extracting greater concurrency than conventional approaches (based on locking data-items) by allowing interleavings not permitted by conventional approaches.

We start by characterizing non-linearizable interleavings permitted by our earlier approach. We classify the interleavings based on the natures of inconsistencies they cause. For each class of interleavings, we describe an extension to our approach to generate additional concurrency control to prohibit these

interleavings. Finally, we prove correctness of our approach by showing that all interleavings we permit are linearizable.

Delayed Falsification The first issue we address, as well as the solution we adopt, are not surprising from a conventional perspective. Informally, the problem with the `Increment` example can be characterized as "dirty reads" and "lost updates": the second procedure invocation reads the original value of `x`, instead of the value produced by the the first procedure invocation; dually, the update done by the first procedure invocation is lost, when the second procedure invocation updates `x`. From a logical perspective, the second invocation relies on the invariant $x == x^{in}$ early on, and the first invocation breaks this invariant later on when it assigns to `x`. This prevents us from reordering the execution to construct an equivalent sequential execution (while preserving the proof). To achieve linearizability, we need to avoid such "delayed falsification".

The extension we now describe prohibits such interference by generating concurrency control to ensure instructions that may falsify predicates and occur after the linearization point will appear to execute atomically along with the linearization point. We achieve this by modifying the strategy to acquire write locks as follows.

- We generalize the earlier notion of *may-falsify*. We say that a path *may-falsify* a predicate φ if some edge in the path may-falsify φ. We say that a predicate φ *may-be-falsified-after* vertex u if there exists some path from u to the exit vertex of the procedure that does not contain the linearization point and may-falsify φ.
- Let $\mathfrak{mf}$ be a predicate map such that for any vertex u, $\mathfrak{mf}(u)$ includes any predicate that may-be-falsified-after u.
- We generalize the original scheme for acquiring write locks. We augment every edge $e = u \xrightarrow{S} v$ as follows:
  1. For every lock ℓ in $\mathfrak{lm}(\mathfrak{mf}(v))\backslash\mathfrak{lm}(\mathfrak{mf}(u))$, we add an "`acquire(`$\ell$`)`" (write lock) before S
  2. For every lock ℓ in $\mathfrak{lm}(\mathfrak{mf}(u))\backslash\mathfrak{lm}(\mathfrak{mf}(v))$, we add an "`release(`$\ell$`)`" (write lock) after S

This extension suffices to produce a linearizable implementation of the example in Figure 3.

Altering Control Flow One interesting aspect of our scheme is that it permits interference that alters the control flow of a procedure invocation if it does not cause the invocation to violate its specification. Such interference may seem benign, but it can lead to non-linearizable behavior. Consider procedures `ReduceX` and `IncY` shown in Figure 5. The specification of `ReduceX` is that it will produce a final state where $x < y$, while the specification of `IncY` is that it

will increment the value of y by 1. `ReduceX` meets its specification by setting x to be $y-1$, but does so *only if* $x \geq y$. Initially, $x = y = 0$.

Now consider a client that invokes `ReduceX` and `IncY` concurrently. The problematic interleaving is the following. Assume that the `ReduceX` invocation enters the procedure. Then, the invocation of `IncY` executes completely. The `ReduceX` invocation continues, and does nothing since $x < y$ at this point. This interleaving produces the final state $x = 0, y = 1$, This is, however, not consistent with a sequential execution where `ReduceX` executes first, followed by an execution of `IncY`, which should produce a final state where $x < y - 1$. (The interleaving is consistent with a sequential execution of the procedures in the opposite order; however, we can enrich the example with other statements that force the sequential execution to be in the order `ReduceX; IncY`.)

Figure 5 also shows a sequential proof and the concurrency control derived by the scheme so far, assuming that the linearization points are at the procedure entry. A key point to note is that `ReduceX`'s proof needs only the single predicate $x < y$. The statement $y = y + 1$ in `IncY` does *not falsify* the predicate $x < y$; hence, `IncY` does not acquire the lock for this predicate. We can characterize the interference in this example as *positive interference*: `IncY` does something that helps, rather than hinders, `ReduceX`. Unfortunately, this means that when we try to linearize the execution, `ReduceX` does not meet its obligation.

We avoid such problems by ensuring that interference by concurrent threads cannot affect the execution path one thread takes. We achieve this by strengthening the notion of positive basis we use as follows: (a) The set of basis predicates at a branch node must be sufficient to express the assume conditions on outgoing edges using disjunctions and conjunctions over the basis predicates, and (b) The set of basis predicates at neighbouring vertices must be positively consistent with each other: for any edge $u \xrightarrow{s} v$, and any predicate φ in the basis at v, the weakest-pre-condition of φ with respect to s must be expressible using disjunctions and conjunctions of the basis predicates at u.

In the current example, this requires the predicate $x \geq y$ to be added to the basis for `ReduceX`. As a result, `ResultX` will acquire lock $l_{x \geq y}$ at entry, while `IncY` will acquire the same lock at its linearization point and release the lock after the statement $y = y + 1$. Again, it is easy to see that the resulting implementation is linearizable.

Affecting Return Values There is still one hurdle we must overcome for linearizability. The extensions so far ensure that interference will not affect a procedure invocation's ability to meet its specification. However, it is still possible for interference to affect *the actual value returned by a procedure invocation*, leading to non-linearizable executions.

Consider procedures `IncX` and `IncY` in Figure 4, which increment variables `x` and `y` respectively. Both procedures return the values of `x` *and* `y`. However, the postconditions of `IncX` (and `IncY`) do not *specify anything about the final value of* `y` (and `x` respectively). Let us assume that the linearization points of the procedures are their entry points. Initially, we have $x = y = 0$.

Consider the following interleaving of a concurrent execution of the two procedures. The two procedures execute the increments in some order, producing the state with $x = y = 1$. Then, both procedures return $(1, 1)$. This execution is non-linearizable because in any legal sequential execution, the procedure executing second is obliged to return a value that differs from the value returned by the procedure executing first.

The left column in Figure 4 shows the concurrency control derived using our approach with the previously described extensions. This is insufficient to prevent the above interleaving. The reason this interference is allowed is that the specification for `IncX` allows it to change the value of `y` arbitrarily; hence, a concurrent modification to `y` by any other procedure is not seen as a hindrance to `IncX`.

Our next extension is designed to prohibit such interference. To do this within our framework, we need to determine whether the execution of a statement s can potentially affect the return-value of another procedure invocation. We will do this by computing a predicate $\phi(ret')$ at every program point u that captures the relation between the program state at point u and the value returned by the procedure invocation eventually (denoted by ret'). We can then check if the execution of a statement s will break predicate $\phi(ret')$, treating ret' as a free variable, to determine if the statement could affect the return value of some other procedure invocation.

Formally, we assume that each procedure returns the value of a special variable ret. (Thus, "`return` exp" is shorthand for "$ret = exp$.) We introduce a special primed variable ret'. We compute a predicate $\phi(u)$ at every program point u such that (a) $\phi(u) =$ `ret'==` ret for the exit vertex u, and (b) for every edge $u \xrightarrow{s} v$, $\{\phi(u)\}s\{\phi(v)\}$ is a valid Hoare triple. In this computation, ret' is treated as a free variable. In effect, this is a weakest-precondition computation of the predicate `ret'==` ret from the exit vertex.

Next, we augment the basis at every vertex u so that it includes a basis for $\phi(u)$ as well. We now apply our earlier algorithm using this enriched basis set.

The middle column in Figure 4 shows the augmented sequential proof of correctness of `IncX`. The concurrency control derived using our approach starting with this proof is shown in Figure 4. The lock l_{merged} denotes a lock obtained by merging locks corresponding to multiple predicates simultaneously acquired/released. It is easy to see that this implementation is linearizable. Also note that if the shared variables `y` and `x` were *not* returned by procedures `IncX` and `IncY` respectively, we will derive a locking scheme in which accesses to `x` and `y` are protected by different locks, allowing these procedures to execute concurrently.

5.4 Correctness

We now present a proof that with the above extensions, our approach guarantees linearizability.

Theorem 2. *Given a library $\mathcal{L}$ that is totally correct with respect to a given sequential specification, the library $\widehat{\mathcal{L}}$ generated by our algorithm is linearizable*

with respect to the given specification.

Proof. Consider any interleaved execution produced by a schedule ξ. We assume, without loss of generality, that every procedure invocation is executed by a distinct thread. Let $t_1, \ldots, t_k$ denote the set of threads which complete execution in the given schedule, ordered so that t_i executes its linearization point before t_{i+1}. Let $\ell_0, \ell_1, \ldots, \ell_n$ represent steps in the schedule ξ, where ℓ_j is an ordered pair (t, e) representing the execution of edge e by thread $t \in \{t_1, \ldots, t_k\}$. We use the notation $t(\ell_j)$ to represent the thread executing ℓ_j, $e(\ell_j)$ to represent the statement executed at ℓ_j and $u(\ell_j)$ to represent the source vertex of the statement $e(\ell_j)$. Let σ_0 denote the initial state. We show that ξ is linearizable by showing that ξ is equivalent to a sequential execution of the specifications of the threads $t_1, \ldots, t_k$ executed in that order.

Let ξ_i denote a *projection* of schedule ξ consisting only of execution steps by threads in the set $\{ t_1, \cdots, t_i \}$.

Lemma 2. *ξ_i is a feasible schedule.*

Proof. We prove the lemma by contradiction. Assume that ξ_i is not feasible. Let ℓ_j be the first infeasible step in ξ_i. Note that ξ_i is obtained from a feasible schedule ξ by omitting steps executed by some threads. Thus, ℓ_j must correspond to some branch: i.e., ℓ_j must contain a statement $assume(\phi)$ such that ϕ is true when ℓ_j is executed in ξ, while ϕ is false when ℓ_j is executed in ξ_i. But our extension for linearizability prevents exactly this kind of interference (by ensuring that one thread's actions cannot alter the control flow of another thread), as shown below.

Let $u_0 \xrightarrow{s_1} u_1 \cdots u_k$ be the path executed by thread $t(\ell_j)$ prior to infeasible step ℓ_j. By construction, there exist sets of predicates $X_p \subseteq \mu(u_p)$, for $1 \leq p \leq k$, such that each X_p is *sufficient to ensure* X_{p+1} in the sense explained below (for $1 \leq p < k$) and $X_k = \{\phi, \neg\phi\}$. We say that X_p is sufficient to ensure X_{p+1} if for every ψ in X_{p+1}, the weakest-precondition of ψ with respect to s_p can be expressed using predicates in X_p using conjunctions and disjunctions.

Our locking scheme ensures that $t(\ell_j)$ holds locks on all predicates in X_p when it is at u_p. This implies that if ϕ is false prior to step ℓ_j in execution ξ_i, then the extra steps performed by other threads in execution ξ cannot make ϕ true. Hence, step ℓ_j must be infeasible in ξ as well, which is a contradiction. □

Next, we show that the projected schedules ξ_i preserve the return values of procedure invocations.

Lemma 3. *The value returned by t_i in the execution of ξ_i is the same as the value returned by t_i in the execution of ξ.*

Proof. This lemma follows from the extension that prevents a procedure from affecting the value returned by another invocation. In a manner similar to the proof of lemma 2, we can show that this extension ensures that any step s_w from threads $(t_{i+1}, \ldots, t_k)$ that can affect the return value of t_i, appears to execute

after the linearization point of t_i. Furthermore, any steps in t_i that affect the return value appear to occur with or before the linearization point of t_i. □

Let σ_i be the state obtained by executing schedule ξ_i from initial state σ_0. Let Φ_i denote the specification for the procedure executed by thread t_i. Consider the sequence of states $\mathcal{S} = \sigma_0, \sigma_1, \ldots, \sigma_k$. The following holds for every pair of states (σ_{i-1}, σ_i).

Lemma 4. $((\sigma_{i-1}, \sigma_i), t_i) \models_c \Phi_i$

Proof. Let σ_i^{lp} represent the program state at the linearization point of thread t_i. From Theorem 1, we know that $((\sigma_i^{lp}, \sigma_i), t_i) \models_c \Phi_i$.

Consider any step ℓ such that $t(\ell) \neq t_i$ and $e(\ell)$ updates shared state. If no such step exists, the lemma trivially holds. Let ϕ be a predicate that $e(\ell)$ may break. If there exists any step ℓ' from thread t_i such that $\phi \in u(\ell')$, then our algorithm would generate concurrency control in t_i to prevent interference from ℓ while ϕ is required to hold in t_i. It follows that such a step ℓ could only have executed before the linearization point of t_i (since the linearization point of t_i occurs after the linearization point of $t_j, 0 \leq j < i$ in ξ). Since none of the basis predicates of any step in t_i are falsified, the lemma holds. □

Note that the sequence $\mathcal{S}$ represents the sequence of intermediate states of a sequential execution of the specifications of the library. This proves that any concurrent schedule ξ of $\widehat{\mathcal{L}}$ is equivalent (corresponding procedures return the same values) to a sequential execution of the specifications. □

The above theorem requires *total correctness* of the library in the sequential setting. *E.g.*, consider a procedure P with a specification `ensures x==0`. An implementation that sets `x` to be 1, and then enters an infinite loop is *partially* correct with respect to this specification (but not totally correct). In a concurrent setting, this can lead to non-linearizable behavior, since another concurrent thread can observe that `x` has value 1, which is not a legally observable value *after* procedure P completes execution.

6 Implementation

We have built a prototype implementation of our algorithm. Our implementation takes a sequential library as input. A pre-processing phase transforms each procedure of the library into a valid C program containing just that procedure (renamed to `main`), its assertions and all global variables. In the first phase of our analysis, we use an existing predicate-abstraction-based software verification tool, adapted to emit a proof of correctness for each of the programs. We compute a positive basis from the proof. A subsequent phase identifies points at which a given procedure breaks predicates in the basis of other procedures. A third phase implements an algorithm for detecting and eliminating deadlocks in the locking scheme and emits the final concurrency control.

Library	Description
compute.c$^+$	See Figure 1
increment.c$^+$	See Figure 5
reduce.c$^+$	See Figure 4
average.c$^+$	Two procedures that compute the sum and average of a set of numbers
device_cache.c	One procedure that reads data from a device and caches the data for subsequent reads [6]. The specification requires quantified predicates.

Table 1: Examples used in our evaluation.

Table 1 shows some of the benchmark programs we have experimented with. Our implementation could automatically generate a proof and derive concurrency control automatically for libraries marked with +. We were restricted to a small class of programs due to limitations of the software model checker in handling complex assertions in libraries that use arrays, pointers and dynamic memory allocation. We have also manually applied our technique to a few libraries that could not be proved correct automatically. For example, we generated a proof of correctness for device cache library in [6] and used the proof to derive the same concurrency control scheme described in the paper.

7 Related Work

Most existing work [8, 4, 7, 14, 12, 16] on synthesizing concurrency control focuses on automatically inferring lock-based synchronization for atomic sections to guarantee atomicity. Linearizability relative to a sequential specification, which we pursue, is a weaker requirement that permits greater concurrency than the notion of atomic sections. Furthermore, existing lock inference schemes for atomic sections identify potential conflicts between atomic section at the granularity of data items and acquire locks to prevent these conflicts, either all at once or a two-phase locking approach. Our approach is novel in using a logical notion of interference (based on predicates), which can potentially permit more concurrency. Finally, the locking disciplines we infer do not necessarily follow two-phase locking, yet guarantee linearizability.

Vechev *et al.* [17] address the problem of automatically deriving linearizable objects with fine-grained concurrency, using hardware primitives to achieve atomicity. The approach is semi-automated, and requires the developer to provide algorithm schema and insightful manual transformations as its first step.

Synthesis and Repair of Concurrent Programs: Early work in [1, 2] synthesizes individual processes in a concurrent program from a detailed multi-

process CTL specification. [15] is a partial synthesis technique that adds missing synchronization by iteratively exploring the space of candidate programs for a given thread schedule, and pruning the search space based on counterexample candidates. [13] uses a model-checking based approach to repair errors in a concurrent program by pruning erroneous paths from the control-flow graph of the interleaved program execution. In [18], the key goal is to obtain a maximally concurrent program for a given cost. This is achieved by deleting transitions from the state-space based on observational equivalence between states, and inspecting if the resulting program satisfies the specification and is implementable. [5] allows users to specify synchronization patterns for critical sections, which are used to infer appropriate synchronization for each of the user-identified region.

8 Extensions and Future Work

Our work opens up a number of interesting ideas and problems that appear worth pursuing.

Optimistic Locking of Library Invariants: A library typically has invariants characterizing its stable state. (These are invariants associated with the quiescent point in our proof). In our scheme, a procedure P holds on to a lock corresponding to a predicate p if it relies on p later in the computation. This approach may be pessimistic for library invariants, since any other procedure that breaks this property is guaranteed to eventually reestablish the invariant again. An alternative scheme, in such a case, would be for P to release the lock on p when it does not immediately require it but acquire it again when needed. Dually, any procedure would acquire the lock before breaking the property and release it only after reestablishing it.

Choosing Good Solutions: This paper presents a space of valid locking solutions that guarantee the desired properties. Specifically, the locking solution generated is dependent on several factors: the sequential proof used, the basis used for the proof, the mapping from basis predicates to locks, the linearization point used, etc. Given a metric on solutions, generating a good solution according to the given metric is a direction for future work. E.g., one possibility is to evaluate the performance of candidate solutions (suggested by our framework) using a suitable test suite to choose the best one.

Fine-Grained Locking: It would be interesting to explore generalizing our approach to infer fine-grained locking. (A fine-grained locking scheme uses an unbounded number of locks, e.g., one associated with each element of a linked list.)

Modular Linearizability Proofs: It would be worth exploring the use of ideas presented here to derive more modular proofs of linearizability of hand-crafted (e.g., lock-free) concurrent data structures, for either automatic or semi-automatic verification.

References

[1] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent systems with many similar processes. In *ACM Transactions on Programming Languages and Systems*, pages 51–115, 1989.

[2] Paul C. Attie and E. Allen Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. volume 23, pages 187–242, 2001.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[4] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proc. of PLDI*, 2008.

[5] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. of ICSE*, pages 442–452. ACM Press, 2002.

[6] Tyfun Elmas, Serdar Tasiran, and Shaz Qadeer. A calculus of atomic sections. In *Proc. of POPL*, 2009.

[7] Michael Emmi, Jeff Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proc. of POPL*, 2007.

[8] Cormac Flanagan and Stephen N. Freund. Automatic synchronization correction. In *Proceedings of the Conference on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005. https://urresearch.rochester.edu/handle/1802/2083.

[9] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: A new algorithm for property checking. In *Proc. of FSE*, November 2006.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL*, pages 58–70. ACM, 2002.

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

[12] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *First Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[13] Muhammad Umar Janjua and Alan Mycroft. Automatic correcting transformations for safety property violations. In *Proceedings of Thread Verification*, pages 111–116, 2006.

[14] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: Synchronization inference for atomic sections. In *Proc. of POPL*, 2006.

[15] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. volume 43, pages 136–148, New York, NY, USA, 2008. ACM.

[16] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proc. of POPL*, pages 334–345, New York, NY, USA, 2006. ACM.

[17] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. volume 43, pages 125–135, New York, NY, USA, 2008. ACM.

[18] Martin Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *IBM Research Report RC24661*, 2008.

```
1 global int x, y;
2 //@ensures x = x^in + 1 ∧ returns (x, y)
3 IncX() {
4   acquire(l_(x==x^in));
5   x = x + 1;
6   (ret11, ret12)=(x, y); //returns(x, y)
7   release(l_(x==x^in));
8 }
9 //@ensures y = y^in + 1&returns (x, y)
10 IncY() {
11   acquire(l_(y==y^in));
12   y = y + 1;
13   (ret11, ret12)=(x, y); //returns(x, y)
14   release(l_(y==y^in));
15 }
```

```
global int x, y;
//@ensures x = x^in + 1 ∧ returns (x, y)
IncX() {
  [ret'11 == x + 1 ∧ ret'12 == y]
  LP : x = x^in
  [x == x^in ∧ ret'11 == x + 1 ∧ ret'12 = y]
  x = x + 1;
  [x == x^in + 1 ∧ ret'11 == x ∧ ret'12 = y]
  (ret11, ret12)=(x, y);
  [x == x^in + 1 ∧ ret11 == ret'11
  ∧ ret12 = ret'12]
}
```

```
global int x, y;
//@ensures x = x' + 1 ∧ returns (x, y)
IncX() {
  acquire(l_merged);
  x = x +1;
  (ret11, ret12)=(x, y);
  release(l_merged);
}
//@ensures y = y' + 1 ∧ returns (x, y)
IncY() {
  acquire(l_merged);
  y = y + 1;
  (ret11, ret12)=(x, y);
  release(l_merged);
}
```

Figure 4: An example that illustrates return value interference. Here, ret_{ij} refers to the j^{th} return variable of the i^{th} procedure.

```
1  global int x, y;
2  //@ensures y = y^in + 1
3  IncY() {
4    acquire(l_{y==y^in})
5    [true]
6    LP : y^in = y
7    [y == y^in]
8    y = y + 1;
9    [y = y^in + 1]
10   release(l_{y==y^in})
11 }
```

```
1  //@ensures x < y
2  ReduceX() {
3    acquire(l_{x<y})
4    [true]
5    LP
6    [true]
7    if (x ≥ y) {
8      [true]
9      x = y - 1;
10   }
11   [x < y]
12   release(l_{x<y})
13 }
```

Figure 5: An example illustrating interference in control flow.