Logical Concurrency Control from Sequential Proofs

Jyotirmoy Deshmukh[1], G. Ramalingam[2], Venkatesh-Prasad Ranganath[2], and Kapil Vaswani[2]

[1] Univeristy of Texas at Austin
jyotirmoy@cerc.utexas.edu
[2] Microsoft Research, India
{grama,rvprasad,kapilv}@microsoft.com

Abstract. We are interested in identifying and enforcing the *isolation requirements* of a concurrent program, i.e., concurrency control that ensures that the program meets its specification. The thesis of this paper is that this can be done systematically starting from a sequential proof, i.e., a proof of correctness of the program in the absence of concurrent interleavings. We illustrate our thesis by presenting a solution to the problem of making a sequential library thread-safe for concurrent clients. We consider a sequential library annotated with assertions along with a proof that these assertions hold in a sequential execution. We show how we can use the proof to derive concurrency control that ensures that any execution of the library methods, when invoked by concurrent clients, satisfies the same assertions. We also present an extension to guarantee that the library is linearizable with respect to its sequential specification.

1 Introduction

A key challenge in concurrent programming is identifying and enforcing the *isolation requirements* of a program: determining what constitutes undesirable interference between different threads and implementing concurrency control mechanisms that prevent this. In this paper, we show how a solution to this problem can be obtained systematically from a *sequential proof*: a proof that the program satisfies a specification in the absence of concurrent interleaving.

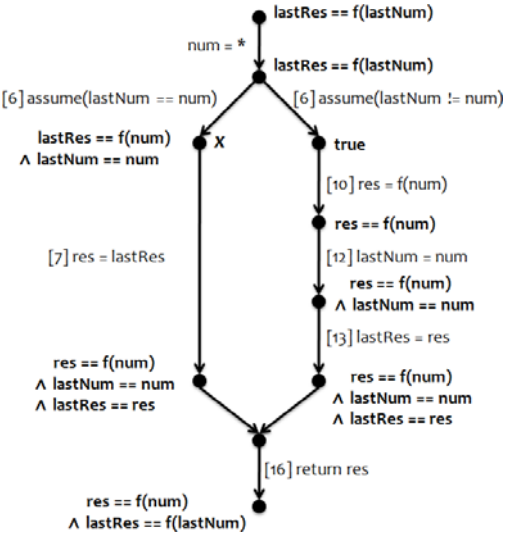
Problem Setting. We illustrate our thesis by considering the concrete problem of making a sequential library safe for concurrent clients. Informally, given a sequential library that works correctly when invoked by any sequential client, we show how to synthesize concurrency control code for the library that ensures that it will work correctly when invoked by any concurrent client.

Consider the example in Fig. 1(a). The library consists of one procedure `Compute`, which applies an expensive function f to an input variable `num`. As a performance optimization, the implementation caches the last input and result. If the current input matches the last input, the last computed result is returned.

A.D. Gordon (Ed.): ESOP 2010, LNCS 6012, pp. 226–245, 2010.

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

(a)



(b)

Fig. 1. (a) (excluding Lines 5,9,11,15) shows a procedure `Compute` that applies a (side-effect free) function f to a parameter `num` and caches the result for later invocations. Lines 5,9,11,15 contain a lock-based concurrency control generated by our technique. (b) shows the control-flow graph of `Compute`, its edges labeled by statements of `Compute` and nodes labeled by proof assertions.

This procedure works correctly when used by a sequential client, but not in the presence of concurrent procedure invocations. E.g., consider an invocation of `Compute(5)` followed by concurrent invocations of `Compute(5)` and `Compute(7)`. Assume that the second invocation of `Compute(5)` evaluates the condition in Line 6, and proceeds to Line 7. Assume a context switch occurs at this point, and the invocation of `Compute(7)` executes completely, overwriting `lastRes` in Line 13. Now, when the invocation of `Compute(5)` resumes, it will erroneously return the (changed) value of `lastRes`.

In this paper, we present a technique that can detect the potential for such interference and synthesize concurrency control to prevent the same. The (lock-based) solution synthesized by our technique for the above example is shown (as comments) in Lines 5, 9, 11, and 15 in Fig. 1(a). With this concurrency control, the example works correctly even for concurrent procedure invocations while permitting threads to perform the expensive function f concurrently.

The Formal Problem. Formally, we assume that the correctness criterion for the library is specified as a set of assertions and that the library satisfies these assertions in any execution of any sequential client. *Our goal is to ensure that any execution of the library with any concurrent client also satisfies the given assertions.* For our running example in Fig. 1(a), Line 3 specifies the desired functionality for procedure `Compute`: `Compute` returns the value f(`num`).

Logical Concurrency Control From Proofs. A key challenge in coming up with concurrency control is determining what interleavings between threads are safe. A conservative solution may reduce concurrency by preventing correct interleavings. An aggressive solution may enable more concurrency but introduce bugs.

The fundamental thesis we explore is the following: a proof that a code fragment satisfies certain assertions in a sequential execution precisely identifies the properties relied on by the code 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 it for our running example. Fig. 1(b) 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. (The statement "`num = *`" at the entry edge indicates that the initial value of parameter `num` is unknown.) A proof consists of an invariant $\mu(u)$ attached 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 $\mu(v)$, (b) The invariant $\mu(entry)$ attached to the entry vertex is satisfied by the initial state and is implied by the invariant $\mu(exit)$ attached to the exit vertex, and (c) for every edge $u \rightarrow v$ annotated with an assertion φ, we have $\mu(u) \Rightarrow \varphi$. Condition (b) ensures that the proof is valid over any *sequence of* executions of the procedure.

The invariant $\mu(u)$ at vertex u indicates the property required (by the proof) to hold at u to ensure that a sequential execution satisfies all assertions of the library. 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 Fig. 1(b). The vertex labeled x in the figure corresponds to the point before the execution of Line 7. The invariant attached to x indicates that the proof of correctness depends on the condition $lastRes{==}f(num)$ being true at x. The execution of Line 10 by another thread will not invalidate this condition. But, the execution of Line 13 by another thread can potentially invalidate this condition. Thus, we infer that, when one thread is at point x, an execution of Line 13 by another thread should be avoided.

We prevent the execution of a statement s by one thread when another thread is at a program point u (if s might invalidate a predicate p that is required at u) as follows. We introduce a lock ℓ_p corresponding to p, and ensure that every thread holds ℓ_p at u and ensure that every thread holds ℓ_p when executing s.

Our algorithm does this as follows. From the invariant $\mu(u)$ at vertex u, we compute a set of predicates $\mathfrak{pm}(u)$. (For now, think of $\mu(u)$ as the conjunction of predicates in $\mathfrak{pm}(u)$.) $\mathfrak{pm}(u)$ represents the set of predicates required at u. For any edge $u \rightarrow v$, any predicate p that is in $\mathfrak{pm}(v) \setminus \mathfrak{pm}(u)$ 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. As a special case, we acquire (release) the locks for all predicates in $\mathtt{pm}(u)$ at procedure entry (exit) when u is the procedure entry (exit) vertex. Finally, if the execution of the statement on edge $u \rightarrow v$ can invalidate a predicate p that is required at some vertex, 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 lead to deadlocks by merging locks when necessary, as described later. Finally, we optimize the synthesized solution using a few simple techniques. E.g., in our example whenever the lock corresponding to `lastRes == res` is held, the lock for `lastNum == num` is also held. Hence, the first lock is redundant and can be eliminated.

Fig. 1 shows the resulting library with the concurrency control we synthesize. This implementation satisfies its specification even in a concurrrent setting. The synthesized solution permits a high degree to concurrency since it allows multiple threads to compute f concurrently. A more conservative but correct locking scheme would hold the lock during the entire procedure execution.

A distinguishing aspect of our algorithm is that it requires only local reasoning and not reasoning about interleaved executions, as is common with many analyses of concurrent programs. Note that the synthesized solution depends on the proof used. Different proofs can potentially yield different concurrency control solutions (all correct, but with potentially different performance).

Linearizability. The above approach can be used to ensure that concurrent executions guarantee desired safety properties, preserve data-structure invariants, and meet specifications (e.g., given as a precondition/postcondition pair). Library implementors may, however, wish to provide the stronger guarantee of linearizability with respect to the sequential specification: *any concurrent execution of a procedure is guaranteed to satisfy its specification and appears 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.

Implementation. We have implemented our algorithm, using an existing software model checker to generate the sequential proofs. We used the tool to successfully synthesize concurrency control for several small examples. The synthesized solutions are equivalent to those an expert programmer would use.

Contributions

We present a technique for synthesizing concurrency control for a library (e.g., developed for use by a single-threaded client) to make it safe for use by concurrent clients. However, we believe that the key idea we present – a technique for identifying and realizing isolation requirements from a sequential proof – can be used in other contexts as well (e.g., in the context of a whole program consisting of multiple threads, each with its own assertions and sequential proofs).

Sometimes, a library designer may choose to delegate the responsibility for concurrency control to the clients of the library and not make the library

thread-safe[1]. Alternatively, library implementers could choose to make the execution of a library method appear atomic by wrapping it in a transaction and executing it in an STM (assuming this is feasible). These are valid options but orthogonal to the point of this paper. Typically, a program is a software stack, with each level serving as a library. Passing the buck, with regards to concurrency control, has to stop somewhere. Somewhere in the stack, the developer needs to decide what degree of isolation is required by the program; otherwise, we would end up with a program consisting of multiple threads where we require every thread's execution to appear atomic, which could be rather severe and restrict concurrency needlessly. The ideas in this paper provide a systematic method for determining the isolation requirements. While we illustrate the idea in a simplified setting, it should ideally be used at the appropriate level of the software stack.

In practice, full specifications are rarely available. We believe that our technique can be used even with lightweight specifications or in the absence of specifications. Consider our example in Fig. 1. A symbolic analysis of this library, with a harness representing a sequential client making an arbitrary sequence of calls to the library, can, in principle, infer that the returned value equals `f(num)`. As the returned value is the only observable value, this is the strongest functional specification a user can write. Our tool can be used with such an inferred specification as well.

Logical interference. Existing concurrency control mechanisms (both pessimistic as well as optimistic) rely on a data-access based notion of interference: concurrent accesses to the same data, where at least one access is a write, is conservatively treated as interfence. A contribution of this paper is that it introduces a more logical/semantic notion of interference that 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" for performance reasons, suggesting that programmers do rely on such a logical notion of interference.

2 The Problem

In this section, we introduce required terminology and formally define the problem. Rather than restrict ourselves to a specific syntax for programs and assertions, we will treat them abstractly, assuming only that they can be given a semantics as indicated below, which is fairly standard.

2.1 The Sequential Setting

Sequential Libraries. A library $\mathcal{L}$ is a pair $(\mathcal{P}, V_G)$, where $\mathcal{P}$ is a set of procedures (defined below), and V_G is a set of variables, termed *global* variables, accessible

[1] This may be a valid design option in some cases. However, in examples such as our running example, this could be a bad idea.

to all and only procedures in $\mathcal{P}$. A procedure P is a pair (G_P, V_P), where G_P is a control-flow graph with each edge labeled by a primitive statement, and V_P is a set of variables, referred to as *local* variables, restricted to the scope of P. (Note that V_P includes the formal parameters of P as well.) To simplify the semantics, we will assume that the set V_P is the same for all procedures and denote it V_L.

Every control-flow graph has a unique entry vertex N_P (with no predecessors) and a unique exit vertex X_P (with no successors). Primitive statements are either `skip` statements, assignment statements, `assume` statements, `return` statements, or `assert` statements. An `assume` statement is used to implement conditional control flow as usual. Given control-flow graph nodes u and v, we denote an edge from u to v, labeled with a primitive statement s, by $u \xrightarrow{s} v$.

To reason about all possible sequences of invocations of the library's procedures, we define the *control graph* of a library to be 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. 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. We refer to these as *call edges*.

Sequential States. A procedure-local state $\sigma_\ell \in \Sigma^s_\ell$ is a pair (pc, σ_d) where pc, the program counter, is a vertex in the control graph and σ_d is a map from the local variables V_L to their values. A global state $\sigma_g \in \Sigma^s_g$ is a map from global variables V_G to their values. A library state σ is a pair $(\sigma_\ell, \sigma_g) \in \Sigma^s_\ell \times \Sigma^s_g$. 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. We assume a standard semantics for primitive statements that 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 labeling) edge e transforms state σ to σ'. 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 $\sigma \overset{e}{\rightsquigarrow}_s \sigma'$.

A *sequential execution* is 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 calls to the library's procedures (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 $i < h \leq j$.

Sequential Assertions. We use `assert` statements to specify desired correctness properties of the library. `Assert` statements have no effect on the execution

semantics and are equivalent to `skip` statements in the semantics. Assertions are used only to define the notion of *well-behaved* executions as follows.

An `assert` statement is of the form `assert` θ where, θ is a 1-state assertion φ or a 2-state assertion Φ. A 1-state assertion, which we also refer to as a predicate, makes an assertion about a single library state. Rather than define a specific syntax for assertions, we assume that the semantics of assertions are defined by a relation $\sigma \models_s \varphi$ denoting that a state σ satisfies the assertion φ.

1-state assertions can be used to specify the invariants expected at certain program points. In general, specifications for procedures take the form of two-state assertions, which relate the input state to output state. We use 2-state assertions for this purpose. The semantics of a 2-state assertion Φ is assumed to be defined by a relation $(\sigma_{in}, \sigma_{out}) \models_s \Phi$ (meaning that state σ_{out} satisfies assertion Φ with respect to state σ_{in}). In our examples, we use special input variables v^{in} to refer to the value of the variable v 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.

Definition 1. *A sequential execution is said to satisfy the library's assertions if for any transition* $\sigma_i \overset{\texttt{assert}\ \theta}{\rightsquigarrow_s} \sigma_{i+1}$ *in the execution, we have (a)* $\sigma_i \models_s \theta$ *if* θ *is a 1-state assertion, and (b)* $(\sigma_{in}, \sigma_i) \models_s \theta$ *where* σ_{in} *is the corresponding entry state of* σ_i*, otherwise. A sequential library satisfies its specifications if every execution of the library satisfies its specifications.*

2.2 The Concurrent Setting

Concurrent Libraries. A concurrent library $\mathcal{L}$ is a triple $(\mathcal{P}, V_G, Lk)$, where $\mathcal{P}$ is a set of concurrent procedures, V_G is a set of global variables, and Lk is a set of global 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 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. 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_ℓ^s. (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.) Let $\Sigma_\ell^c = T \to \Sigma_\ell^s$ denote the set of all local states.

At any point during execution, a lock *lk* is either free or held by one thread. We represent the state of locks by a partial function from Lk to T indicating which thread, if any, holds any given lock. Let $\Sigma_{\text{lk}}^c = Lk \hookrightarrow T$ represent the set of all lock-states. Let $\Sigma_g^c = \Sigma_g^s \times \Sigma_{\text{lk}}^c$ denote the set of all global states. Let $\Sigma^c = \Sigma_\ell^c \times \Sigma_g^c$ denote the set of all states. 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. The concurrent semantics is induced by the sequential semantics as follows. 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 $\sigma \overset{(t,e)}{\rightsquigarrow}_c \sigma'$.

A *concurrent execution* is 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 $0 \leq i < k$. We say that $\ell_0 \cdots \ell_{k-1}$ is the *schedule* of this execution. A sequence $\ell_0 \cdots \ell_m$ 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 $i < h \leq j$.

We note that our semantics uses sequential consistency. Extending our results to support weaker memory models is future work.

Interpreting Assertions In Concurrent Executions. 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 a 1-state assertion φ in the context of thread t_i (denoted by $(\sigma, t_i) \models_c \varphi$) iff $\sigma[t_i] \models_s \varphi$. For any 2-state assertion 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 2. *A concurrent execution π is said to satisfy the library's assertions if for any transition $\sigma_i \overset{(t,\texttt{assert}\ \theta)}{\rightsquigarrow}_c \sigma_{i+1}$ in the execution we have (a) $(\sigma_i, t) \models_c \theta$, if θ is a 1-state assertion, and (b) $((\sigma_{in}, \sigma_i), t) \models_c \theta$ where σ_{in} is the corresponding t-entry state of σ_i, otherwise. 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` and a procedure `IncX` that increments `x` by 1. A possible specification for `IncX` 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`. Explicitly stating such frame conditions is unnecessarily restrictive, as a concurrent update to `y` by another procedure, when `IncX` is executing, would be considered a violation of `IncX`'s specification. Frame conditions can be handled better by treating a specification as a pair (S, Φ) where S is the set of all global variables referenced by 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 $(\{x\}, x == x^{in} + 1))$. In the sequel, however, we will restrict ourselves to the simpler setting and ignore this issue.

2.3 Goals

Our goal is: Given a sequential library $\mathcal{L}$ with assertions satisfied 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. 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

A *sequential proof* is a mapping μ from vertices of the control graph to predicates such that (a) for every edge $e = u \xrightarrow{t} v$, $\{\mu(u)\}t\{\mu(v)\}$ is a valid Hoare triple (i.e., $\sigma_1 \models_s \mu(u)$ and $\sigma_1 \overset{e}{\rightsquigarrow}_s \sigma_2$ implies $\sigma_2 \models_s \mu(v)$), and (b) for every edge $u \xrightarrow{\textit{assert } \varphi} v$, 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. The preceding paragraph, however, hints at the interference we must avoid to ensure correctness: 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 break this invariant. 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 elaborate on this idea below.

A *predicate mapping* is a mapping $\mathfrak{pm}$ from the vertices of the control graph to a set of predicates. A predicate mapping $\mathfrak{pm}$ is said to be a *basis* for a proof μ if every $\mu(u)$ can be expressed as a boolean formula (involving conjunctions, disjunctions, and negation) over $\mathfrak{pm}(u)$. A basis $\mathfrak{pm}$ for proof μ is *positive* if every $\mu(u)$ can be expressed as a boolean formula involving only conjunctions and disjunctions over $\mathfrak{pm}(u)$.

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. However, we want to formalize 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 straightforward as these can be handled just like extra program variables.)

E.g., consider Line 13 in Fig. 1. Consider predicate $lastRes{==}f(num)$. By renaming local variable `num` to avoid naming conflicts, we obtain predicate `lastRes ==f(nûm)`. We say that Line 13 *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 $\mathtt{pm}$ be a positive basis for a proof μ and $\mathcal{R} = \cup_u \mathtt{pm}(u)$. If a predicate φ is in $\mathtt{pm}(u)$, 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 $\varphi \in \mathtt{pm}(u)$. 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 $\mathtt{pm}(v) \setminus \mathtt{pm}(u)$. We say that predicate φ *becomes relevant* at edge e. In the motivating example, the predicate `lastNum == num` becomes relevant at Line 12

We ensure the desired invariant by acquiring the locks corresponding to every predicate that becomes relevant at edge e *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 falsify predicate φ.)

Now consider any predicate φ in $\mathtt{pm}(u) \setminus \mathtt{pm}(v)$. We say that φ *becomes irrelevant* at edge e. E.g., predicate `lastres == f(lastNum)` becomes irrelevant once the false branch at Line 8 is taken. For every p that becomes irrelevant at edge e, we release 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 $\varphi \in \mathtt{pm}(u)$), and add a release of the same lock after s (unless $\varphi \in \mathtt{pm}(v)$).

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 $u \xrightarrow{return} v$ must be augmented to release all locks corresponding to predicates in $\mathtt{pm}(u)$ before returning. Dually, any procedure entry edge $w \rightarrow u$ must be augmented to acquire all locks corresponding to predicates in $\mathtt{pm}(u)$.

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 $e = u \to v$: (a) if e may falsify a library invariant φ, then φ must be in $\mathfrak{om}(v)$, and (b) if $\varphi \in \mathfrak{om}(u)$, then φ must be in $\mathfrak{om}(v)$ unless e *establishes* φ. Here, we say that an edge $u \xrightarrow{s} v$ *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.* Any lock to be acquired along an `assume` edge will need to be acquired before the condition is evaluated. If the lock is not required along all `assume` edges out of a vertex, then we will have to release the lock along the edges 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 $u \xrightarrow{s} v$ such that $p \in \mathfrak{m}(u) \wedge r \in (\mathfrak{m}(v) \cup \mathfrak{mbf}(u \xrightarrow{s} v)) \setminus \mathfrak{m}(u)$. 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. We map all predicates in an equivalence class to the same lock to 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. 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, reader-writer locks can be used to improve concurrency, but space constraints prevent a discussion of this extension.

Generating Proofs. The sequential proof required by our scheme can be generated using verification tools such as SLAM [2], BLAST [10,11] and Yogi [9]. Since a minimal proof can lead to better concurrency control, approaches that produce a "parsimonious proof" (e.g., see [11]) are preferable. A parsimonious proof is one that avoids the use of unnecessary predicates at any program point.

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 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 $\mathfrak{m}(u) = \mathfrak{pm}(u) \cup \mathfrak{om}(u)$:

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

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) $\forall\ p \in \mathit{BasisLocksAcq}(u \xrightarrow{s} v)$, add an `acquire(`$\mathfrak{lm}(p)$`)` before s;
   (b) $\forall\ p \in \mathit{BasisLocksRel}(u \xrightarrow{s} v)$, add a `release(`$\mathfrak{lm}(p)$`)` after s;
   (c) $\forall\ p \in \mathit{BreakLocks}(u \xrightarrow{s} v)$, 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

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.

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

See [5] for all proofs.

4 Extensions for 2-State Assertions

The algorithm presented in the previous section can be extended to handle 2-state assertions via a simple program transformation that allows us to treat 2-state assertions (in the original program) as single-state assertions (in the transformed program). 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 original variable v to the corresponding new variable $\tilde{v}$.

Let $\underline{\sigma'}$ denote the projection of a transformed program state σ' to a state of the original program obtained by forgetting the values of the new variables. Given a 2-state assertion Φ, let $\tilde{\Phi}$ denote the single-state assertion 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 original program corresponds to satisfaction of the single-state assertion $\tilde{\Phi}$ in the transformed program.

Lemma 1. *(a) A schedule ξ is feasible in the transformed program iff it is feasible in the original program. (b) Let σ' and σ be the states produced by a particular schedule with the transformed and original programs, respectively. Then, $\sigma = \underline{\sigma'}$. (c) Let π' and π be the executions produced by a particular schedule with the transformed and original program, 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 $\tilde{\Phi}$.*

Synthesizing concurrency control. We now apply the technique discussed in Section 3 to the transformed program to synthesize concurrency control that preserves the assertions transformed as discussed above. It follows from the above Lemma that this concurrency control, used with the original program, preserves both single-state and two-state assertions.

5 Guaranteeing Linearizability

In the previous section, we showed how to 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 and preventing compositional reasoning in clients of the library. E.g., consider the procedure `Increment` shown in Fig. 2, 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 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. (We note that such concerns do not arise

```
1 int x = 0;
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 }
```

Fig. 2. A non-linearizable implementation of the procedure `Increment`

when the specification does not refer to shared variables. For instance, the specification for our example in Fig. 1 does not refer to shared variables, even though the implementation uses shared variables.)

One solution to this problem is to apply concurrency control synthesis to the code (library) that calls `Increment`. The synthesis can then detect the potential for interference between the calls to `Increment` and prevent them from happening concurrently. Another possible solution, which we explore in this section, is for the library to guarantee a stronger correctness criteria called *linearizability* [12]. Linearizability gives the illusion that in any concurrent execution, (the sequential specification of) every procedure of the library appears to execute *instantaneously* at some point between its call and return. This illusion allows clients to reason about the behavior of concurrent library compositionally using its sequential specifications.

In this section, we extend our approach to derive concurrency control that guarantees linearizability. Due to space constraints, we show how to ensure that every procedure appears to execute instantaneously along its entry edge, while satisfying its sequential specification. The technique can be generalized to permit linearization points (i.e., the point at which the procedure'appears to execute instantaneously) other than the procedure entry, subject to some constraints (see [5]). Recall that we adapt the control-flow graph representation of each procedure by labelling the procedure entry edge with the statement $\mathcal{LP}$ defined in Section 4 to handle 2-state assertions. Without loss of generality, we assume that each procedure P_j returns the value of a special local variable ret_j.

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

Delayed Falsification. Informally, the problem with the `Increment` example can be characterized as "dirty reads" and "lost updates": the second procedure invocation executes its linearization point later than the first procedure invocation but reads the original value of `x`, instead of the value produced by the the first 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` (at a point when the second invocation no longer relies on the invariant). This prevents us from reordering the execution to construct an equivalent sequential execution (while preserving the proof).

The extension we now describe prevents such interference by ensuring that instructions that may falsify predicates and occur after the linearization point appear to execute atomically at the linearization point. We achieve this by modifying the strategy to acquire 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 any 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 locks. We augment every edge $e = u \xrightarrow{S} v$ as follows:
  1. $\forall\ \ell\ \in\ \mathfrak{lm}(\mathfrak{mf}(v))\backslash\mathfrak{lm}(\mathfrak{mf}(u))$, add an "`acquire(`$\ell$`)`" before S
  2. $\forall\ \ell \in\ \mathfrak{lm}(\mathfrak{mf}(u))\backslash\mathfrak{lm}(\mathfrak{mf}(v))$, add an "`release(`$\ell$`)`" after S

This extension suffices to produce a linearizable implementation of the example in Fig. 2.

Return Value Interference. We now focus on interference that can affect *the actual value returned by a procedure invocation*, leading to non-linearizable executions.

Consider procedures `IncX` and `IncY` in Fig. 5, 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 Fig. 5 shows the concurrency control derived using our approach with previously described extensions. This is insufficient to prevent the above interleaving. This interference is allowed because 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`.

To prohibit such interferences 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 do this by computing a predicate $\phi(ret')$ at

```
int x, y;
IncX() {
  acquire(l_{x==x^in});
  x = x + 1;
  (ret_11,ret_12)=(x,y);
  release(l_{x==x^in});
}
IncY() {
  acquire(l_{y==y^in});
  y = y + 1;
  (ret_21,ret_22)=(x,y);
  release(l_{y==y^in});
}
```

(a)

```
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]
  (ret_11,ret_12)=(x,y);
[x==x^in+1 ∧ ret_11==ret'_11
∧ ret_12==ret'_12]
}
```

(b)

```
int x, y;
IncX() {
  acquire(l_merged);
  x = x+1;
  (ret_11,ret_12)=(x,y);
  release(l_merged);
}
IncY() {
  acquire(l_merged);
  y = y+1;
  (ret_21,ret_22)=(x,y);
  release(l_merged);
}
```

(c)

Fig. 3. An example illustrating return value interference. Both procedures return `(x,y)`. ret_{ij} refers to the j^{th} return variable of the i^{th} procedure. Figure 3(a) is a non-linearizable implementation synthesized using the approach described in Section 3. Figure 3(b) shows the extended proof of correctness of the procedure `IncX` and Figure 3(c) shows the linearizable implementation.

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 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 Fig. 5 shows the augmented sequential proof of correctness of `IncX`. The concurrency control derived using our approach starting with this proof is shown in the third column of Fig. 5. 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.

Control Flow Interference. An interesting aspect of our scheme is that it permits interference that alters the control flow of a procedure invocation if it does not

```
1 int x, y;
2 //@ensures y = y^in + 1
3 IncY() {
4       [true] LP : y^in = y
5       [y == y^in] y = y + 1;
6       [y == y^in + 1]
7 }
```

```
1 //@ensures x < y
2 ReduceX() {
3       [true] LP
4       [true] if (x ≥ y) {
5       [true] x = y - 1;
6                   }
7       [x < y]
8 }
```

Fig. 4. An example illustrating interference in control flow. Each line is annotated (in square braces) with a predicate the holds at that program point.

cause the invocation to violate its specification. Consider procedures `ReduceX` and `IncY` shown in Fig. 4. 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* $x \geq y$.

Now consider a client that invokes `ReduceX` and `IncY` concurrently from a state where $x = y = 0$. 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.

Fig. 4 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. This locking scheme permits `IncY` to execute concurrently with `ReduceX` and affect its control flow. While our approach guarantees that this control flow interference will not cause assertion violations, proving linearizability in the presence of such control flow interference, in the general case, is challenging (and an open problem). Therefore, we conservatively extend our scheme to prevent control flow interference, which suffices to guarantee linearizability.

We ensure that interference by one thread does not affect the execution path another thread takes. We achieve this by strengthening the notion of positive basis 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 predicate $x \geq y$ to be added to the basis for `ReduceX`. As a result, `ReduceX` 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 $y = y + 1$. It is easy to see that this implementation is linearizable.

Correctness. The extensions described above to the algorithm of Sections 3 and 4 for synthesizing concurrency control are sufficient to guarantee linearizability, as stated in the theorem below.

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.*

6 Implementation

We have built a prototype implementation of our algorithm that uses a predicate-abstraction based software verification tool [9] to generate the required proofs. Our implementation takes a sequential library and its assertions as input. It uses a pre-processing phase to combine the library with a harness (that simulates the execution of any possible sequence of library calls) to get a valid C program. It then use the verification tool to generate a proof of correctness for this program. It then uses the algorithm presented in this paper to synthesize concurrency control for the library.

We used a set of benchmark programs to evaluate our approach. The programs include examples shown in Figure 1, 5 and 4. We also used two real world libraries, a device cache library [6] that reads data from a device and caches the data for subsequent reads, and a C implementation of the Simple Authentication and Security Layer (SASL). This library is a generic server side library that manages security context objects for user sessions. We applied our technique manually to the device cache library and the SASL library because the model checker we used does not permit quantifiers in specifications. For these libraries, we wrote full specifications (which required using quantified predicates) and manually generated proofs of correctness.

Starting with these (manually and automatically generated) proofs, the concurrency control scheme we synthesized was identical to what an experienced programmer would generate (in terms of the number and scope of locks). Our solutions permit more concurrency as compared to naive solutions that use one global lock or an atomic section around the body of each procedure. In all cases, the concurrency control scheme we synthesize are the same or better than the concurrency control defined by developers of the library. For example, in case of the server store library, our scheme generates smaller critical sections and identifies a larger number of critical sections that acquire different locks as compared to the default implementation. The source code for all our examples and their concurrent versions are available online at [1]. We leave a more detailed evaluation of our approach as future work.

7 Related Work

Synthesizing Concurrency Control. Most existing work [8,3,7,15,13,19] on synthesizing concurrency control focuses on inferring lock-based synchronization

for atomic sections to guarantee atomicity. Our work differs in expoiting a (sequential) specification to derive concurrency control. We also present an extension to guarantee linearizability with respect to a sequential specification, which is a weaker requirement that permits greater concurrency than the notion of atomic sections. Furthermore, existing lock inference schemes identify potential conflicts between atomic sections at the granularity of data items and acquire locks to prevent these conflicts, either all at once or using a two-phase locking approach. Our approach is novel in using a logical notion of interference (based on predicates), which can permit more concurrency. Finally, the locking disciplines we infer do not necessarily follow two-phase locking, yet guarantee linearizability.

[18] describes a sketching technique to add missing synchronization by iteratively exploring the space of candidate programs for a given thread schedule, and pruning the search space based on counterexample candidates. [14] uses model-checking to repair errors in a concurrent program by pruning erroneous paths from the control-flow graph of the interleaved program execution. In [21], 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. [4] allows users to specify synchronization patterns for critical sections, which are used to infer appropriate synchronization for each of the user-identified region. Vechev *et al.* [20] 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. Our approach differs from all of these techniques in exploiting a proof of correctness (for a sequential computation) to synthesize concurrency control that guarantees thread-safety.

Verifying Concurrent Programs. Our proposed style of reasoning is closely related to the axiomatic approach for proving concurrent programs of Owicki & Gries [17]. While they focus on proving a concurrent program correct, we focus on synthesizing concurrency control. They observe that if two statements *do not interfere*, the Hoare triple for their parallel composition can be obtained from the sequential Hoare triples. Our approach identifies statements that *may interfere* and violate the sequential Hoare triples, and then synthesizes concurrency control to ensure that sequential assertions are preserved by parallel composition.

Prior work on verifying concurrent programs [16] has also shown that attaching invariants to resources (such as locks and semaphores) can enable modular reasoning about concurrent programs. Our paper turns this around: we use sequential proofs (which are modular proofs, but valid only for sequential executions) to identify critical invariants and create locks corresponding to such invariants and augment the program with concurrency control that enables us to lift the sequential proof into a valid proof for the concurrent program.

References

1. WYPIWYG examples (June 2009), `http://research.microsoft.com/en-us/projects/wypiwyg/wypiwyg_examples.zip`
2. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)
3. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: Proc. of PLDI (2008)
4. Deng, X., Dwyer, M.B., Hatcliff, J., Mizuno, M.: Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In: Proc. of ICSE, pp. 442–452 (2002)
5. Deshmukh, J., Ramalingam, G., Ranganath, V.P., Vaswani, K.: Logical concurrency control from sequential proofs. Tech. Rep. MSR-TR-2009-81, Microsoft Research (2009)
6. Elmas, T., Tasiran, S., Qadeer, S.: A calculus of atomic sections. In: Proc. of POPL (2009)
7. Emmi, M., Fischer, J., Jhala, R., Majumdar, R.: Lock allocation. In: Proc. of POPL (2007)
8. Flanagan, C., Freund, S.N.: Automatic synchronization correction. In: Proc. of SCOOL (2005)
9. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Proc. of FSE (November 2006)
10. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. of POPL, pp. 58–70 (2002)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. of POPL, pp. 232–244 (2004)
12. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. Proc. of ACM TOPLAS 12(3), 463–492 (1990)
13. Hicks, M., Foster, J.S., Pratikakis, P.: Lock inference for atomic sections. In: First Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (2006)
14. Janjua, M.U., Mycroft, A.: Automatic correcting transformations for safety property violations. In: Proc. of Thread Verification, pp. 111–116 (2006)
15. McCloskey, B., Zhou, F., Gay, D., Brewer, E.A.: Autolocker: Synchronization inference for atomic sections. In: Proc. of POPL (2006)
16. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
17. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. In: Proc. of CACM (1976)
18. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: Proc. of PLDI, pp. 136–148 (2008)
19. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: Proc. of POPL, pp. 334–345 (2006)
20. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Proc. of PLDI, pp. 125–135 (2008)
21. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 139–154. Springer, Heidelberg (2009)