Synthesizing Fine-Grained Synchronization Protocols for Implicit Monitors

KOSTAS FERLES*, The University of Texas at Austin, USA
BENJAMIN SEPANSKI*, The University of Texas at Austin, USA
RAHUL KRISHNAN, The University of Texas at Austin, USA
JAMES BORNHOLT, The University of Texas at Austin, USA
ISIL DILLIG, The University of Texas at Austin, USA

A *monitor* is a widely-used concurrent programming abstraction that encapsulates all shared state between threads. Monitors can be classified as being either *implicit* or *explicit* depending on the primitives they provide. Implicit monitors are much easier to program but typically not as efficient. To address this gap, there has been recent research on automatically synthesizing explicit-signal monitors from an implicit specification [Ferles et al. 2018], but prior work does not exploit all paralellization opportunities due to the use of a *single* lock for the entire monitor. This paper presents a new technique for synthesizing *fine-grained* explicit-synchronization protocols from implicit monitors. Our method is based on two key innovations: First, we present a new static analysis for inferring *safe interleavings* that allow violating mutual exclusion of monitor operations *without* changing its semantics. Second, we use the results of this static analysis to generate a MaxSAT instance whose models correspond to correct-by-construction synchronization protocols. We have implemented our approach in a tool called Cortado and evaluate it on monitors that contain parallelization opportunities. Our evaluation shows that Cortado can synthesize synchronization policies that are competitive with, or even better than, expert-written ones on these benchmarks.

1 INTRODUCTION

Concurrent programming is difficult because it requires developers to consider interactions between multiple threads of execution and mediate access to shared resources and data. Programming languages can offer higher-level abstractions to reduce this complexity by making concurrent programming more declarative. One such abstraction is the *monitor* [Hansen 1973; Hoare 1974], which is an object that encapsulates shared state and allows threads access to it only through a set of *operations*, between which the monitor enforces mutual exclusion.

Ideally, developers would implement monitors using *implicit synchronization*, wherein the *only* synchronization primitive is a `waituntil(P)` operation that blocks threads until condition `P` is satisfied. The compiler or runtime can then automatically generate the necessary *explicit synchronization* operations (locks, condition variables, etc.) to implement the monitor in a way that respects the semantics of the implicit monitor. However, automatically deriving an efficient explicit monitor from its implicit specification is a challenging problem, and there have been several recent research efforts, including both run-time techniques like AutoSynch [Hung and Garg 2013] and compile-time tools like Expresso [Ferles et al. 2018], to support implicit-synchronization monitors.

While these state-of-the-art approaches make it possible to program using implicit monitors, they still achieve sub-optimal performance because they adhere closely to the monitor's mutual exclusion

*Both authors contributed equally to the paper.

Authors' addresses: Kostas Ferles, Computer Science Department, The University of Texas at Austin, USA, kferles@cs.utexas.edu; Benjamin Sepanski, Computer Science Department, The University of Texas at Austin, USA, ben_sepanski@utexas.edu; Rahul Krishnan, Computer Science Department, The University of Texas at Austin, USA, rahulk@cs.utexas.edu; James Bornholt, Computer Science Department, The University of Texas at Austin, USA, bornholt@cs.utexas.edu; Isil Dillig, Computer Science Department, The University of Texas at Austin, USA, isil@cs.utexas.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

requirement. They generally use a *single* lock for the entire monitor and allow access by *at most one* thread at a time across all monitor operations. In practice, however, many monitors can admit additional concurrency while still preserving the *appearance* of mutual exclusion. For example, consider a FIFO queue monitor that provides `take` and `put` operations. These two operations can safely run concurrently unless the queue is empty or full, as they will not access the same slot in the queue. Today, realizing this *fine-grained* concurrency requires expert developers to fall back to hand-written explicit synchronization. These implementations are subtle and error-prone, and there is no easy way for developers to determine when they have extracted the maximum possible concurrency from such an implementation.

This paper presents a new technique to automatically synthesize fine-grained explicit-synchronization monitors. Our technique takes as input an implicit monitor that specifies the desired operations and automatically generates an implementation that allows as much concurrency as possible between those operations while still preserving the appearance of mutual exclusion. The key idea is to decompose each monitor operation into a set of *fragments* and allocate a *set* of locks to each fragment to enforce the mutual exclusion requirement while allowing as many fragments as possible to run concurrently. The resulting implementation selectively acquires and releases locks at fragment boundaries within each operation and signals condition variables as needed.

At a high level, our approach operates in three phases to generate a high-performance explicit synchronization monitor from its implicit version:

- **Signal placement:** First, we use an off-the-shelf technique [Ferles et al. 2018] to infer a signaling regime which determines where to insert signaling operations on condition variables. While the output of this tool is sufficient to synthesize a single-lock implementation, it does not admit any additional concurrency wherein different threads can perform monitor operations simultaneously.
- **Static analysis:** Second, we perform static analysis to infer sufficient conditions for correctness. That is, the output of the static analysis is a set of conditions such that if the synthesized monitor obeys them, it is guaranteed to be correct-by-construction. A key challenge for this static analysis is to determine which fragments can safely execute concurrently without creating a potential violation of the monitor semantics. The analysis simulates *interleaving* each fragment between the fragments of other operations and determines which possible interleavings are safe.
- **Synchronization protocol synthesis via MaxSAT:** Finally, we reduce the synthesis problem to a maximum satisfiability (MaxSAT) instance from whose solution an explicit sychronization protocol can be extracted. The hard constraints in the MaxSAT problem enforce the correctness requirements extracted by the static analysis, while the soft constraints encode two competing objective functions: minimizing the total number of locks used, while maximizing the number of pairs of fragments that can run concurrently.

We have implemented our proposed approach in a tool called Cortado that operates on Java monitors and evaluated it on a collection of monitor implementations that are (1) drawn from popular open-source projects and (2) contain parallelization opportunities that can be achieved via fine-grained locking. Given only the implicit monitor as input, Cortado synthesizes explicit-synchronization monitors that perform as well as, or better than, hand-written explicit implementations by expert developers. Compared to state-of-the-art automated tools for synthesizing explicit monitors [Ferles et al. 2018], Cortado-synthesized monitors extract more concurrency and therefore perform much better (up to 39.1×) on heavily contended workloads.

In summary, this paper makes four main contributions:

- A new technique for automatically synthesizing fine-grained monitor implementations that admit the maximum possible concurrency.
- A novel static static analysis for inferring safe interleaving opportunities between threads.

```
class ArrayBlockingQueue {
  int first = 0, last = 0, count = 0;
  Object[] queue;

  ArrayBlockingQueue(int capacity) {
    if (capacity < 1)
      throw new IllegalArgumentException();
    this.queue = new Object[capacity];
  }

  void put(Object o) {
    // Fragment 1
    waituntil(count < queue.length);
    // Fragment 2
    queue[last] = o;
    // Fragment 3
    last = (last + 1) % queue.length;
    // Fragment 4
    count++;
  }

  Object take() {
    // Fragment 5
    waituntil(count > 0);
    // Fragment 6
    Object r = queue[first];
    queue[first] = null;
    // Fragment 7
    first = (first + 1) % queue.length;
    // Fragment 8
    count--;
    return r;
  }
}
```

(a) Implicit-synchronization ArrayBlockingQueue.

```
class ArrayBlockingQueue {
  int first = 0, last = 0; Object[] queue;
  AtomicInteger count = new AtomicInteger(0);

  Lock putLock = new Lock(), takeLock = new Lock();
  Condition notFull = putLock.newCondition();
  Condition notEmpty = takeLock.newCondition();
  // Constructor is the same as the implicit version.

  void put(Object o) {
    putLock.lock()
    while (count.get() == queue.length)
      notFull.await();
    queue[last] = o;
    last = (last + 1) % queue.length;
    int c = count.getAndIncrement();
    putLock.unlock();
    if (c == 0) {
      takeLock.lock();
      notEmpty.signalAll();
      takeLock.unlock();}}

  Object take() {
    takeLock.lock();
    while (count.get() == 0)
      notEmpty.await();
    Object r = queue[first];
    queue[first] = null;
    first = (first + 1) % queue.length;
    int c = count.getAndDecrement();
    takeLock.unlock();
    if (c == queue.length) {
      putLock.lock();
      notFull.signalAll();
      putLock.unlock();}
    return r;}}
```

(b) Explicit-synchronization ArrayBlockingQueue.

Fig. 1. Motivating example.

- A MaxSAT encoding to automate reasoning about *both* the correctness and performance of the synthesized explicit-synchronization monitor.
- An implementation of our technique, CORTADO, that outperforms both state-of-the-art automated tools and expert-written code on benchmarks that can be parallelized via fine-grained locking.

2 OVERVIEW

In this section, we give an overview of our approach through a motivating example. Given the implicit-synchronization monitor shown in Figure 1a, our goal is to automatically synthesize an efficient and semantically equivalent explicit-synchronization monitor like the one presented in Figure 1b. In what follows, we walk through this example and describe how our technique is able to automatically generate the code in Figure 1b.

2.1 Implicit-synchronization monitor

Our technique takes as input an *implicit-synchronization monitor* that specifies which operations should execute atomically and when certain operations are allowed to proceed but does not fix a specific synchronization protocol for realizing that behavior. For example, Figure 1a shows an implicit monitor that implements a limited capacity blocking queue via a bounded circular array buffer. This monitor defines two operations, `put` and `take`, that execute atomically (i.e., the body of each method must appear to execute as one indivisible unit). The `put` operation adds an object if the queue is not full, and `take` removes an object if the queue is not empty. If one of these method calls cannot proceed (i.e., queue is full or empty), the monitor blocks the calling thread's execution using

a `waituntil` statement until the operation can be executed. For example, the `waituntil` statement at line 13 in `take` blocks execution until there is at least one object in the queue.

As Figure 1a illustrates, implicit-synchronization monitors make concurrent programming simpler because they are *declarative*: they merely state which operations are atomic and when operations can proceed, but they do *not* specify a particular synchronization protocol for realizing that desired behavior. However, most programming languages do not offer implicit synchronization facilities; so, concurrent programs must instead be implemented in terms of *explicit* synchronization constructs such as locks and condition variables, as we discuss next.

2.2 Explicit-synchronization monitor

Figure 1b shows an explicit-synchronization implementation of the bounded queue from Figure 1a that is written by an expert. This implementation uses two distinct locks, `putLock` and `takeLock`, to protect the `put` and `take` methods respectively. The explicit-synchronization monitor also uses an atomic integer for the `count` field, transforming reads into `get()` calls (e.g., line 12) and writes into the appropriate atomic method (e.g., `count.getAndIncrement()` on line 16). The expert-written monitor performs explicit signaling via condition variables `notFull` and `notEmpty` that are associated with `putLock` and `takeLock` respectively. When a thread cannot execute one of these operations, it calls `await` on the appropriate condition variable to block its execution (lines 13 and 26). A thread blocked in `put` can only be unblocked by a corresponding `take` that frees up space in the queue. To do so, the `take` must acquire `putLock` and perform a signal operation on condition variable `notFull` (lines 33–35). The logic for `take` is symmetric (lines 19–21).

Although the expert-written version has more locks than a single global-lock implementation, its performance will often be better: Introducing two locks allows `put` and `take` to execute concurrently, although multiple concurrent `puts` are still serialized, as are multiple `takes`. A single global lock would admit no concurrency in this case and would still incur the same synchronization overhead of acquiring and releasing a lock on every method call. The expert implementation mitigates the overhead of having two locks by acquiring locks selectively: `take` only acquires the `putLock` if it is possible for there to be a `put` operation currently blocked waiting for space in the queue, which happens only if the queue was full when `take` ran (the `put`/`takeLock` case is symmetric). This example demonstrates the intricacy of synthesizing fine-grained locking protocols: instead of only minimizing the total number of locks, we must also try to maximize the available concurrency.

2.3 Our Approach

Our tool Cortado automatically synthesizes the efficient explicit-synchronization monitor in Figure 1b given the implicit version from Figure 1a. It does so in three phases: First, it infers when and how signaling operations should take place. Second, it performs static analysis to infer sufficient conditions for the synthesized monitor to be correct. Third, it encodes the synchronization protocol synthesis problem as a MaxSAT instance and uses a model of the MaxSAT problem to generate an explicit-sychronization monitor. Since prior work can already handle the first phase, we only focus on the the latter two phases in the following discussion.

Granularity. The granularity of our synthesized locking protocol is at the level of *code fragments*, where each fragment is a single-entry region of code within a single method. For example, the fragments chosen for the blocking queue example are indicated by comments in Figure 1a. Fragments are the indivisible unit of concurrency in our approach: we aim to maximize the number of fragments that can run concurrently, but we do not modify the code within a fragment to introduce extra concurrency (e.g., by removing data races). Hence, the explicit monitor synthesized by our approach acquires and releases locks only at fragment boundaries.

Static analysis. To ensure correctness of the synthesized monitor, our technique needs to enforce the following three key requirements:

(1) **Data-race freedom:** Fragments that involve a data race must not be able to run concurrently.
(2) **Deadlock freedom:** Locks must be acquired and released in an order that prevents deadlocks.
(3) **Atomicity:** Each monitor operation should *appear* to take place as one indivisible unit. That is, even though the implementation can allow thread interleavings inside monitor operations, the resulting state should be equivalent to one where each method executes truly atomically.

Here, the second requirement (i.e., deadlock freedom) does not necessitate any static analysis, as we can prevent deadlocks by imposing a static total order $\preceq$ on locks [Birrell 1989] and ensuring that locks are acquired and released in a manner that is consistent with $\preceq$. However, in order to ensure data-race freedom and atomicity, we need to perform static analysis of the source code to identify (1) code fragments that have a data race, and (2) interleaving opportunities between code fragments. Since detection of data races is a well-studied problem, the novelty of our static analysis lies in identifying safe interleaving opportunities. Hence, the key question addressed by our analysis is the following: Given a code fragment f executed by thread t, and two consecutive code fragments f_1, f_2 executed by a different thread t', is it safe to interleave the execution of f *in between* f_1 and f_2 while ensuring that monitor operations *appear* to take place atomically?

To answer this question, our method performs a novel static analysis to identify a set of such *safe interleavings*. For instance, going back to the running example, our analysis determines that it is safe to interleave the execution of fragment 4 in Figure 1a in between fragments 5 and 6 by checking a number of commutativity relations between code fragments. In this instance, since our analysis proves that fragment 4 left-commutes [Lipton 1975] with fragment 5 and right-commutes [Lipton 1975] with 6 and all of its successors, we identify this as a safe interleaving opportunity. On the other hand, our analysis concludes that interleaving fragment 4 in between 1 and 2 is *not* safe because fragment 4 does not left-commute with fragment 1 — intuitively, this is because fragment 4 can falsify predicate `count < queue.length` that appears in the `waituntil` statement of fragment 1.

MaxSAT overview. Once we identify possible data races and safe interleavings via static analysis, we use this information to generate a MaxSAT instance whose solution corresponds to a fine-grained synchronization protocol. Specifically, our MaxSAT encoding uses a variable $h_{f_i}^{l_j}$ to indicate that code fragment f_i must hold lock l_j and generates both hard constraints (for correctness) and soft constraints (for efficiency) over these variables. Thus, if the MaxSAT solver returns a model in which variable $h_{f_i}^{l_j}$ is assigned to true, this means that the synthesized code must acquire lock l_j prior to executing fragment f_i. Similarly, our MaxSAT encoding introduces a variable a_{fld} indicating that field *fld* should be implemented using an atomic type.

The hard constraints in our MaxSAT encoding correspond to the three correctness requirement mentioned earlier, namely (1) data race prevention, (2) deadlock freedom, and (3) atomicity. On the other hand, soft constraints encode our optimization objective. In what follows, we give a brief overview of the different types of constraints in our encoding, focusing only on constraints that involve lock acquisition variables $h_{f_i}^{l_j}$. However, it is worth noting that our technique also generates constraints on atomic variables a_{fld} and can automatically convert fields to atomic types whenever doing so is safe and more efficient than introducing a lock.

Data-race freedom. Given a pair of code fragments (f_i, f_j) that have a potential data race according to the static analysis, our MaxSAT encoding introduces hard constraints of the form $\bigvee_k (h_{f_i}^{l_k} \wedge h_{f_j}^{l_k})$ stating that f_i and f_j must share at least one common lock. For example, in Figure 1a, our analysis determines that fragments 4 and 8 cannot run in parallel since they both write to the

same memory location `count`. Thus, the MaxSAT instance contains boolean constraints to make sure that two different threads cannot execute `count--` and `count++` at the same time.

Deadlock freedom. Our approach precludes deadlocks by imposing a total order $\preceq$ on locks. In particular, it enforces that a thread t can only acquire lock l if t does *not* already hold any lock l' where $l' \prec l$. For example, in Figure 1a, suppose the locking protocol determines that fragments 1 and 2 must hold all locks in sets L_1 and L_2 respectively. Between executing the two fragments, the code will need to acquire all locks in $L_2 \setminus L_1$. Hence, we add constraints $i < j$ for every pair of locks $l_j \in L_2 \setminus L_1$ and $l_i \in L_1 \cap L_2$ so that those locks can be acquired while respecting the order $\preceq$.

Atomicity. Our MaxSAT encoding also includes constraints to ensure that monitor operations appear to execute atomically. Suppose that our static analysis determines that a thread cannot safely execute code fragment f *in between* some other thread's execution of code fragments f_1 and f_2. To prevent such an unsafe interleaving, we add hard constraints to ensure that fragments f, f_1, and f_2 all share at least one common lock. For example, since our analysis determines that fragment 4 (`count++`) cannot be interleaved with any other pair of fragments in the same method `put` (running concurrently on a different thread), our MaxSAT encoding includes a hard constraint asserting that fragment 4 must share a lock with all other fragments in the `put` method.

Soft constraints. Because the efficiency of the synthesized code depends on both the allowed parallelization opportunities as well as the number of locks, our optimization objective tries to *minimize* the number of locks and *maximize* the number of fragments that can run in parallel. To encode the latter objective, our MaxSAT encoding includes soft contraints asserting that any two parallelizable fragments must *not* share a lock. On the other hand, to encode the former objective, we add a soft constraint stating that no fragment in m is holding lock l.

Monitor generation. A solution of the generated MaxSAT instance determines (a) which fragments should hold which locks, (b) which fields should be implemented using atomic types, and (c) which locks should be associated with which condition variables. Thus, together with the output of the signal placement technique [Ferles et al. 2018], a model of the MaxSAT problem can be automatically translated into the target monitor implementation. For our running example, Cortado synthesizes precisely the implementation in Figure 1b given the implicit monitor of Figure 1a.

3 PRELIMINARIES

In this section, we describe our source and target languages and define what it means for an explicit synchronization monitor to correctly implement an implicit one.

3.1 Background on Monitors

In this work, we assume that all shared resources between threads are handled by a *monitor class* M which consists of fields F and set of operations (methods) O. The fields F constitute the *only* shared state between threads, which can only access shared state by performing one of the monitor operations $o \in O$. These operations can be performed by an arbitrary, yet fixed, number of threads, and locations reachable through arguments are assumed to be thread-local. We represent each thread by a unique identifier from set $T \subseteq \mathbb{N}$, and we model memory locations using *access paths* (AP) [Landi and Ryder 1992] of the form $\pi = v(.f)*$, consisting of a base variable v optionally followed by a finite sequence of field accesses. We also assume that a special `this` variable stores the memory location of the monitor object.

Definition 3.1. **(Monitor State)**. A monitor state $\sigma : T \times AP \rightarrow \mathbb{N}$ is a mapping from pairs (t, π) (where t is a thread identifier and π an access path) to a value.

Monitor M	::=	`monitor` M `{`$(fld \mid m)^*$`}`
Field fld	::=	$\tau\ f := e$
Method m	::=	$m(\vec{v})\{ccr^*\}$
CCR ccr	::=	`waituntil`(p); s
Stmt s	::=	$skip \mid v := e \mid v.f := e$
	|	$v.m(\vec{e}) \mid$ `[if` (e)`]? goto` `l`
	|	$ls_1; ls_2$
LStmt ls	::=	`l:?` s

(a) Implicit-synchronization monitor language.

Monitor M	::=	`monitor` M `{`$(fld \mid sync \mid m)^*$`}`
Field fld	::=	$\tau\ f := e$
Sync $sync$	::=	`Lock` l `:= new Lock()`
	|	`CondVar` cv `:=` l`.newCondVar()`
	|	`Atomic`$[\tau]\ a := e$
Method m	::=	$m(\vec{v})$ `{` ccr^* `}`
CCR ccr	::=	$(ls)^*$
Stmt s	::=	$skip \mid v := e \mid v.f := e$
	|	$v.m(\vec{e}) \mid$ `[if` (v)`]? goto` `l`
	|	$ls_1; ls_2$
	|	$a_{pre} := a$`.update(`$\lambda\chi.e$`)`
LStmt ls	::=	`l:?` s

(b) Explicit-synchronization monitor language.

Fig. 2. Source & target languages. We use e and p for expressions and predicates respectively.

3.2 Source Language

Our source language, presented in Figure 2a, corresponds to *implicit synchronization monitors* without explicit locking or signaling. The body of each monitor operation consists of a sequence of so-called *Conditional Critical Regions (CCRs)* [Hoare 1971], which in turn consist of a `waituntil` statement followed by one or more regular non-blocking statements. We refer to the predicate of the `waituntil` statement of a CCR as its *guard* and to the rest of the statements as its *body*. A thread executes the body of the CCR atomically if its guard evaluates to true; otherwise it suspends its execution and exits the monitor until the predicate becomes true. More formally, the semantics of our source language are defined via the notion of an *implicit monitor history*:

Definition 3.2. **(Implicit monitor history).** Given a set of threads interacting with each other through monitor $M_s = (F, O)$, an *implicit monitor history* h_i is a sequence $(ccr_1, t_1) \dots (crr_n, t_n)$ where each ccr_i is a CCR of M_s and t_i is a thread identifier.

Given history h_i, we define an *argument mapping* v_i to be a list whose i'th element maps formal parameters of $Method(ccr_i)$ to their actual value for each event (ccr_i, t_i) in h_i.

Definition 3.3. **(Implicit monitor semantics).** Given a monitor M_s, initial state σ, and monitor history h_i with argument mapping v_i, the operational semantics of M is defined using a judgment $M_s \vdash (h_i, v_i, \sigma) \Downarrow \sigma'$ indicating that the new monitor state is σ' after executing h_i on state σ.

Because our source language is very similar to the one used in Ferles et al. [2018], we omit a formal definition of the operational semantics. Following that work, we also consider an implicit history to be valid only if it respects the program order of the input monitor.

3.3 Target Language

Figure 2b presents the language of explicit-synchronization monitors. The overall structure of this target language is similar to the source language but with a few important differences. First, an explicit monitor contains locks, conditional variables, and atomic fields, collectively referred to as *synchronization variables*. Second, CCRs in the target language do not contain `waituntil` statements; instead, the logic of a `waituntil` statement is implemented by calling methods on the appropriate condition variable. We assume that synchronization variables support all the standard

```
class M {
  int x = 0, y = 0, z = 0;
  void foo() { x++; y++; }
  void bar() { z++; } }
```

(a) A simple implicit monitor.

```
class M {
  int x = 0, y = 0, z = 0;
  Lock l1 = new Lock(), l2 = new Lock();
  void foo() { l1.lock(); x++; y++; l1.unlock(); }
  void bar() { l2.lock(); z++; l2.unlock(); } }
```

(b) An explicit monitor implementation of Figure 3a.

$$h_i = (foo, t_1)(bar, t_2)$$
$$h_e = (\texttt{l1.lock()}, t_1)(\texttt{x++}, t_1)(\texttt{y++}, t_1)(\texttt{l1.unlock()}, t_1)(\texttt{l2.lock()}, t_2)(\texttt{z++}, t_2)(\texttt{l2.unlock()}, t_2)$$
$$h'_e = (\texttt{l1.lock()}, t_1)(\texttt{x++}, t_1)(\texttt{l2.lock()}, t_2)(\texttt{y++}, t_1)(\texttt{z++}, t_2)(\texttt{l1.unlock()}, t_1)(\texttt{l2.unlock()}, t_2)$$

(c) Examples of implicit and explicit histories.

Fig. 3. A simple implicit monitor and its explicit implementation.

synchronization operations present in modern concurrent languages (e.g., `await`, `signal`, `signalAll`, etc.). Finally, our target language contains a special `update` statement for performing updates on atomic fields: it takes as argument an atomic field a and a unary function f and updates the value of a *atomically* as $f(a)$. For instance, the statement $\texttt{c}_{pre}$ `:= c.update(`$\lambda\chi.\chi + 1$`)` atomically increments `c` by one and stores the value of `c` before the update in $\texttt{c}_{pre}$.

Definition 3.4. **(Explicit monitor history)**. Given a set of threads executing in monitor $M_t = (F, O)$, an *explicit monitor history* h_e is a sequence $(s_1, t_1) \dots (s_n, t_n)$ where each s_i is a (non-composite) statement of a monitor operation $o \in O$ and t_i is a thread identifier.

Leveraging the same notion of *argument mappings* defined in Section 3.2, we define explicit monitor semantics as follows:

Definition 3.5. **(Explicit monitor semantics)**. Given a monitor M_t, initial state σ, and monitor history h_e with argument mapping ν_e, the operational semantics of M_t is defined using a judgment $M_t \vdash (h_e, \nu_e, \sigma) \downarrow \sigma'$ indicating that the new state is σ' after executing h_e on initial state σ.

The full operational semantics of our target language is given in Appendix C.

3.4 Relating Implicit and Explicit Histories

In order to formalize the correctness of our approach, we need to relate an implicit history h_i of a source monitor M_s with an explicit history h_e of its corresponding target version M_t. Because every history of an implicit monitor M_s *induces* a corresponding history of its explicit version M_t, we define an operation called that Expand that "translates" an implicit history to an explicit one. That is, given an implicit history h_i with argument mapping ν_i and state σ, $\text{Expand}_{M_t}(h_i, \nu_i, \sigma)$ returns a pair (h_e, ν_e), where h_e is a history of M_t containing all statements executed by h_i under initial state σ and ν_e is the argument mapping for h_e.

Example 3.6. Consider the implicit monitor of Figure 3a and its explicit counterpart in Figure 3b. For histories h_i and h_e from Figure 3c we have $\text{Expand}_{M_t}(h_i, \nu_i, \sigma) = (h_e, \nu_e)$ for some ν_i, ν_e.

Using this Expand operation, we can classify explicit histories as being sequential or interleaved:

Definition 3.7. **(Sequential history)** Let M_t be an explicit monitor implementation of M_s. We say that an explicit history h_e of monitor M_t with argument mapping ν_e is *sequential* iff there exist a history h_i of M_s, argument mapping ν_i, and initial state σ such that $\text{Expand}_{M_t}(h_i, \nu_i, \sigma) = (h_e, \nu_e)$.

In other words, a sequential history corresponds to an execution in which statements of the explicit monitor are not interleaved between threads.

Example 3.8. Going back to Figure 3c, history h_e is sequential but h'_e is not.

Next, we introduce the notion of *well-formed histories*, which, intuitively, respect the program order of the original implicit monitor:

Definition 3.9. **(Well-formed history)** Let $\Pi(h, t)$ be the *projection* of h onto thread t (i.e., it filters out all elements of h not involving thread t). We say that a history h_e of M_t is *well-formed* iff, for every thread t, there exists sequential histories $h_e^1, \ldots, h_e^n$ such that $\Pi(h_e, t) = h_e^1 \cdots h_e^n$.

Intuitively, well-formed histories respect program dependence in the original monitor for every thread. By definition, every sequential history is also well-formed. In the remainder of this paper, we implicitly mean *well-formed* explicit history whenever we refer to an explicit history.

Example 3.10. Histories h_e, h'_e from Figure 3c are both well-formed. However, the following history is not well-formed because it does not respect program order: $(\texttt{l2.unlock()}, t)(\texttt{l2.lock()}, t)$

Definition 3.11. **(Interleaved history)** We say that a history h_e of M_t is *interleaved* iff it is (1) well-formed and (2) not sequential.

Example 3.12. History h'_e from Figure 3c is interleaved.

Next, we define what it means for an explicit history to *simulate* an implicit history.

Definition 3.13. **(Simulation relation).** Let M_t be an explicit version of implicit monitor M_s. We say that an explicit history h_e of M_t with argument mapping ν_e simulates (h_i, ν_i) of M_s on input σ, denoted $(h_e, \nu_e) \backsim (h_i, \nu_i)$, if there exist sequential history h'_e and ν'_e such that:

$$(1)\ \forall t.\ \Pi(h_e, t) = \Pi(h'_e, t) \qquad \textit{and} \qquad (2)\ \mathsf{Expand}_{M_t}(h_i, \nu_i, \sigma) = (h'_e, \nu'_e).$$

In other words, h_e simulates a history of the original monitor if it is a (well-formed) permutation of some sequential history h'_e of the explicit monitor M_t.

Example 3.14. Going back to Figure 3c, we have $(h'_e, \nu') \backsim (h_i, \nu)$ for some ν, ν'.

3.5 Correctness of Explicit-Synchronization Monitors

Using the concepts introduced in the previous section, we now formalize what it means for an explicit monitor to *correctly implement* an implicit one.

Definition 3.15. **(State equivalence)** Let σ be a program state of an implicit monitor M_s and σ' that of an explicit monitor M_t. We say that σ and σ' are equivalent modulo M_s, denoted $\sigma \equiv_{M_s} \sigma'$, iff for all (t, π) in the domain of σ, we have $\sigma(t, \pi) = \sigma'(t, \pi)$

Intuitively, this notion of equivalence between two monitor states ignores any additional synchronization fields and local variables introduced by translating M to an explicit-synchronization monitor. Finally, we can define the correctness of an explicit monitor as follows:

Definition 3.16. **(Correctness)** We say that an explicit monitor M_t correctly implements an implicit monitor M_s, denoted as $M_s \sim M_t$, iff for all input states σ_s, σ_t s.t. $\sigma_s \equiv_{M_s} \sigma_t$, we have:

$$(1)\ \forall h_i, \nu_i.\ M_s \vdash (h_i, \nu_i, \sigma_s) \Downarrow \sigma'_s \implies \left(M_t \vdash (\mathsf{Expand}_{M_t}(h_i, \nu_i, \sigma_s), \sigma_t) \downarrow \sigma'_t \ \wedge\ \sigma'_s \equiv_{M_s} \sigma'_t\right)$$

$$(2)\ \forall h_e, \nu_e.\ M_t \vdash (h_e, \nu_e, \sigma_t) \downarrow \sigma'_t \implies \left(\exists h_i, \nu_i.\ (h_e, \nu_e) \backsim (h_i, \nu_i) \wedge M_s \vdash (h_i, \nu_i, \sigma_s) \Downarrow \sigma'_s \wedge \sigma'_s \equiv_{M_s} \sigma'_t\right)$$

The first correctness condition simply states that M_t does not eliminate any feasible behaviors of M_s. The second condition, on the other hand, states that every feasible history of M_t simulates *some* implicit history that results in the same state. Intuitively, this means that all statement interleavings allowed by M_t provide the illusion that all operations of M_s are executed atomically.

4 MAIN ALGORITHM

In this section, we present our main synthesis algorithm. Specifically, Section 4.1 introduces some preliminary definitions and proves an NP-completeness result to justify the reduction to MaxSAT. Then, Section 4.2 presents the high-level algorithm, Section 4.3 presents the static analysis for inferring safe interleavings, and Sections 4.4 presents the details of the MaxSAT encoding.

4.1 Fragment Dependency Graphs and NP-Completeness

Our main synthesis algorithm is parametrized over a partitioning of the input monitor into code fragments, where each code fragment defines a unit of computation that we need to assign locks to. In this section, we clarify our assumptions about these code fragments and prove the NP-completeness of the problem for a given choice of partition.

First, to define what we mean by a valid partition, we represent each method of the monitor as a standard control-flow graph (CFG), where each atomic statement belongs to its own basic block. Given a control-flow graph G and node n, we write $Preds(G, n)$ to indicate the predecessor nodes of n in G and $Succs(G, n)$ to indicate its successors. Then, a valid partition of a method into code fragments is defined as follows:

Definition 4.1. **(Partition)** Let $G = (V, E)$ be the CFG representation of a method. Then, a partition of this method is a set of CFGs $\{G_1, \ldots, G_n\}$ with $G_i = (V_i, E_i)$ such that:

(1) $V = \uplus_{i=1}^{n} V_i$ and $E_i = E \cap (V_i \times V_i)$
(2) For every G_i, there is at most one node $n \in V_i$ such that $Preds(G, n) \not\subseteq V_i$
(3) Every `waituntil`(p) statement must belong to its own G_i — i.e., if a node $n \in V$ is a waituntil statement, then there exists a $G_i = (\{n\}, \emptyset)$

Intuitively, a *partition* is a set of sub-CFGs such that (1) these sub-CFGs cover all nodes of the original CFG, (2) each sub-CFG has a unique entry node, and (3) waituntil statements belong to their own sub-CFG. We refer to the code snippet represented by each sub-CFG as a *code fragment* and define a notion of *fragment dependency graph (FDG)* as follows:

Definition 4.2. **(FDG)** Given a method m with CFG $G = (V, E)$ and a partition of G into $\{G_1, \ldots, G_n\}$, a *fragment dependency graph (FDG)* is a directed acyclic graph (V', E') such that (1) every $f_i \in V'$ is the code fragment associated with G_i; (2) there is an edge $(f_i, f_j) \in E'$ iff there is an edge in G from any exit node of G_i to the entry node of G_j.

Example 4.3. Figure 4 presents the FDG of method `take` for the partition in Figure 1a,

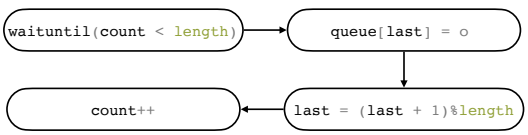

Fig. 4. FDG for method `take`.

Observe that we require the FDG to be acyclic, so some partitions do not give rise to valid FDGs. In the rest of this paper, we assume that partitions obey this restriction so that all cycles are contained within individual nodes of the FDG. We also lift this notion of FDG from individual methods to entire monitors in the obvious way (i.e., union of all FDGs). As we will see in the next section, our synthesis algorithm operates over FDG representations of monitors.

Next, we state the following NP-completeness result to justify our MaxSAT encoding:

Theorem 4.4. **(NP-Completeness)** *Let $\mathcal{G} = (V, E)$ be an FDG of monitor M, and let $\Pi \subseteq V \times V$ be a set of fragment pairs that can run in parallel. Then, deciding whether there exists a synchronization protocol with at most k locks and that allows all pairs in Π to run in parallel is NP-Complete.*

Proof. By reduction from the edge clique cover problem [Michael and Quint 2006]. The proof can be found in Appendix B. □

4.2 Synthesis Algorithm

In this section, we describe our core synthesis procedure, which is summarized in Figure 5. At a high level, the SynthesizeMonitor algorithm consists of the following steps. First, it uses the technique of Ferles et al. [2018] to infer signaling operations (line 4). This yields a partially concretized monitor M' with signaling operations but no locking. Next, it constructs an FDG representation

```
1:  procedure SynthesizeMonitor(M)
2:      input: M: an implicit-synchronization monitor.
3:      output: a semantically equivalent explicit-synchronization monitor.
4:      M' ← PlaceSignals(M)          ▷ Use technique of Ferles et al. to infer signaling operations
5:      G ← ConstructFDG(M')
6:      N_u ← ComputeMaxLocks(G)
7:      S ← StaticAnalyze(G)
8:      opt ← −1
9:      for i ∈ [1, N_u] do
10:         (H, S) ← MaxSatEncoding(M, G, S, i)
11:         (p, v, timeout) ← Solve(H, S)
12:         if timeout ∨ (v ≤ opt) then break
13:         (best, opt) ← (p, v)
14:     return Intrument(best, G, M')
```

Fig. 5. Main Synthesis Algorithm.

of the resulting monitor M' as defined in Section 4.1 (line 5). Third, it infers an *upper bound* $\mathcal{N}_u$ on the maximum number of locks that the synthesized code should use (line 6). Then, it statically analyzes the FDG to infer requirements that the synthesized code needs to obey (line 7) and uses the results of the previous steps to generate the MaxSAT encoding (line 11). Finally, it instruments M' (line 14) using the synchronization protocol inferred using MaxSAT. Since the most involved aspects of this algorithm are the MaxSAT encoding and inference of safe interleavings, we defer a detailed discussion of these to the next two subsections and focus on the rest.

Iterative exploration of lock count. As mentioned above, our synthesis algorithm conceptually reduces the protocol synthesis problem to MaxSAT and uses an off-the-shelf solver to maximize our optimization objective. To achieve this goal, one option is to generate the MaxSAT encoding based on the maximum possible locks (obtained via the call to ComputeMaxLocks) and then let the solver figure out the optimal number of locks to use. However, *in practice*, such an approach does not scale because the size of the encoding increases with respect to the maximum number of locks allowed. That is, for many realistic problems, the MaxSAT solver fails to terminate within a reasonable time limit if we generate the encoding based on the maximum possible locks. Thus, instead of directly generating a very large MaxSAT formula up front, our SynthesizeMonitor procedure enters a loop (lines 9–13) wherein it gradually increases the maximum number of locks allowed (and hence the size of the MaxSAT encoding). If we get to a point where the MaxSAT solver starts timing out (indicated by boolean variable called *timeout*) or we fail to increase the objective value despite using a larger upper bound on locks (see line 12), then the procedure terminates with the best sychronization policy found so far. While this strategy does not guarantee global optimality, it is much more practical than the alternative.

Signaling operations. Our synthesis algorithm uses an auxiliary procedure called PlaceSignals [Ferles et al. 2018] which yields a monitor M' that belongs to an intermediate language that is identical to our source language (Figure 2a) except that it contains explicit signaling operations. Specifically, this intermediate language contains two additional signaling directives: (1) `signal(p,c)` which notifies a single thread that is blocked on predicate `p` if condition `c` holds, and (2) `broadcast(p,c)` which notifies *all* threads blocked on `p` if `c` holds. Figure 6 shows the result of calling PlaceSignals on the `put` procedure from Figure 1a.

```
void put(Object o) {
  waituntil(count < queue.length);
  boolean wasEmpty = count == 0;
  queue[last] = o;
  last = (last + 1) % queue.length;
  count++;
  broadcast(count == 0, wasEmpty);
}
```

Fig. 6. Method put with explicit signals.

FDG construction. Recall that an FDG is a generalized version of a control-flow graph where nodes are code fragments rather than basic blocks, and each code fragment is a unit of computation that our algorithm should assign locks to. Since there can be many ways to partition a given method into code fragments, the ConstructFDG procedure invoked at line 5 of Figure 5 implements a particular heuristic for partitioning a method into code fragments. In particular, the more the number of code fragments, the more the parallelization opportunities; thus, our ConstructFDG procedure tries to maximize the number of code fragments while maintaining the invariant that the FDG is acyclic and that each code fragment must have a unique entry point (see Section 5).

Computing upper bound on locks. Because the MaxSAT encoding assumes a fixed number of locks, our synthesis algorithm calls the ComputeMaxLocks procedure at line 6 to compute an upper bound on the number of locks needed. Given an FDG $\mathcal{G} = (V, G)$, the key idea behind this procedure is to construct a so-called *conflict graph* $G_C = (V, E_C)$ where (f, f') is in E_C iff fragments f and f' have a data race. Since it can be shown that the optimal solution to our problem is an *edge clique cover* [Michael and Quint 2006] of this conflict graph (see Appendix), we can use known theorems (e.g., Mantel's theorem, Alon [1986] etc.) to obtain an upper bound on the number of locks needed without having to solve an NP-complete problem.[1]

Static analysis. Recall from Section 2 that the constraints in our MaxSAT encoding utilize information obtained via static analysis. Thus, line 7 of Figure 5 statically analyzes the input monitor to obtain the following three pieces of information:

- **Atomic fields $\mathcal{F}$:** One of the goals of the analysis is to infer a set of fields that could *potentially* be implemented using `Atomic` types. Thus, our static analysis checks whether (a) a field of type `T` has a corresponding `AtomicT` version, and (b) whether all updates to this field can be implemented using one of the methods provided by `AtomicT`.
- **Data races $\mathcal{R}$:** The second goal of our static analysis is to identify pairs of fragments that would have a data race if they do not use a shared lock. Thus, given a pair of fragments (f, f'), our static analysis checks whether f writes to a memory location l that is accessed in f'.
- **Interleaving opportunities $\mathcal{I}$:** Finally, a third key goal of the analysis is to identify safe interleaving opportunities between fragments. Since this aspect of the analysis is quite involved, we discuss it in the next subsection.

MaxSAT encoding. As mentioned in Section 2, our MaxSAT encoding uses two types of boolean variables, namely (1) $h_{f_i}^{l_j}$ indicating that fragment f_i must hold lock l_j and (2) a_f indicating that field f should be converted to atomic. Hence, a model of the MaxSAT problem can be easily converted to a so-called *locking protocol* $(\mathcal{L}, \mathcal{A}, \mathcal{P})$, where $\mathcal{L}$ is an assignment from fragments to a set of locks, $\mathcal{A}$ is a set of fields that should be implemented using atomic types, and $\mathcal{P}$ is a mapping from

[1] In our implementation, we use multiple upper bounds using known theorems and return the best one.

`waituntil` guards to locks. In particular, we have $l_j \in \mathcal{L}(f_i)$ if and only if $h_{f_i}^{l_j}$ is assigned to true in the model returned by the MaxSAT solver, and we have $fld \in \mathcal{A}$ if a_{fld} is assigned to true. Due to the constraints in our MaxSAT encoding, it is similarly easy to derive $\mathcal{P}$: because our encoding ensures that every occurrence of a `waituntil(p)` statement is protected by the *same* set of locks S, we associate one of the locks l in S with the condition variable introduced for predicate p.[2]

Instrumentation. The last step of our algorithm is to synthesize the explicit-synchronization monitor via the Instrument procedure invoked at line 14. Given a synchronization protocol $(\mathcal{L}, \mathcal{A}, \mathcal{P})$, the Instrument procedure performs the following steps:

(1) First, it introduces all the synchronization fields (locks, condition variables, and atomic fields) that appear in the protocol.
(2) It converts every update to an atomic field to the corresponding atomic `update` statement.
(3) Finally, it introduces all the necessary locking and signaling operations to implement the synthesized synchronization protocol.

We refer the interested reader to Appendix D for more details on the instrumentation.

Theorem 4.5. **(Correctness)** *Given an implicit-synchronization monitor M in the language of Figure 2a, if* SynthesizeMonitor(M) *returns M', then we have $M \sim M'$.*

Proof. Can be found in Appendix E. □

4.3 Analysis to Identify Safe Interleavings

We now describe how to infer safe interleaving opportunities between threads while ensuring that monitor operations *appear* to take place atomically. Given a fragment dependency graph $\mathcal{G} = (V, E)$ for a monitor M, an *interleaving opportunity* (or *interleaving* for short) is a pair (v, e) where $v \in V$ is a code fragment of M and $e = (v_1, v_2) \in E$ is an edge of the FDG. Intuitively, such an interleaving is safe if some thread can execute v *in between* some other thread's execution of v_1 and v_2 without violating atomicity. The goal of our static analysis is to identify a set $\mathcal{I}$ of such safe interleavings. In what follows, we formalize *safe interleavings* and describe an analysis for identifying them.

Formalizing safe interleavings. To formalize the notion of safe interleaving, we need to keep track of which fragments of the monitor were executed in what order. For this purpose, given an FDG $\mathcal{G} = (V, E)$ of M, we define a *fragmented monitor* $M_{\mathcal{G}}$ to be the same as M except that every fragment in $\mathcal{G}$ is placed in its own method. Observe that histories of $M_{\mathcal{G}}$ encode all possible interleavings of fragments in $\mathcal{G}$. In this sense, histories of $M_{\mathcal{G}}$ are similar to explicit monitor histories but are slightly higher level in that they allow interleavings between fragments rather than atomic statements. Thus, we adapt the same notions of *sequential*, *well-formed*, and *interleaved* histories from Section 3.3 to fragmented monitors, as illustrated by the following examples.

Example 4.6. Given monitor M from Figure 1a, its fragmented version $M_{\mathcal{G}}$ splits `put` and `take` into four different methods, each named $\texttt{put}_i$ and $\texttt{take}_i$. Given history $h = (take, t)$ and initial state σ with a non-empty queue, we have

$$\mathsf{Expand}_{M_{\mathcal{G}}}(h, v, \sigma) = ((take_1, t)(take_2, t)(take_3, t)(take_4, t), v')$$

where $take_1, \ldots, take_4$ denote fragments 5-8 in Figure 1a and v,v' are empty argument mappings.

[2]Specifically, when choosing which lock l in set S to designate as the representative, we choose the smallest lock in S according to the total order. Because all locks held by a thread must be released before it blocks on a condition variable and must be acquired after it gets notified (with method `await` releasing and acquiring l internally), choosing the smallest lock prevents deadlocks.

Example 4.7. In the example above, $\text{Expand}_{M_\mathcal{G}}(h, v, \sigma)$ is both sequential and well-formed. However, $(take_1, t), (take_2, t)$ is not well-formed because it does not involve all four methods, and $(take_1, t), (take_3, t), (take_2, t), (take_4, t)$ is also not well-formed because it executes $take_3$ before $take_2$. Finally, the following history is an interleaved (and, by definition, well-formed) history where threads t and t' execute method `take` and `put` respectively:

$$h_\mathcal{G} = (put_1, t)(put_2, t)(put_3, t)(take_1, t')(put_4, t)(take_2, t')(take_3, t')(take_4, t') \tag{1}$$

Furthermore, for this history we have $(h_\mathcal{G}, v_\mathcal{G}) \backsim ((put, t)(take, t'), v)$ for some $v_\mathcal{G}$ and v. That is, $h_\mathcal{G}$ *simulates* a history of M where thread t executes method `put` and t' executes `take`.

Definition 4.8. **(Interleaving)** Given an FDG $\mathcal{G} = (V, E)$ for monitor M, an *interleaving* is a pair (v, e) where $v \in V$ and $e \in E$. Furthermore, given a history h of fragmented monitor $M_\mathcal{G}$, we write $\mathcal{X}(h_\mathcal{G})$ to denote the set of all interleavings that occur in h.

Example 4.9. For the history $h_\mathcal{G}$ from Eq. 1, we have:

$$\mathcal{X}(h_\mathcal{G}) = \{(take_1, (put_3, put_4)), (put_4, (take_1, take_2))\}$$

This is the case because this history executes $take_1$ in between consecutive fragments put_3 and put_4 of some other thread. Similarly, we have $(put_4, (take_1, take_2)) \in \chi(h_\mathcal{G})$ because it executes put_4 in between $take_1$ and $take_2$.

Definition 4.10. **(Safe interleavings).** Let $\mathcal{G}$ be an FDG of monitor M. We say that a set of interleavings S is *safe*, if for every input state σ and every interleaved history $h_\mathcal{G}$ of $M_\mathcal{G}$ we have:

$$\text{If } \mathcal{X}(h_\mathcal{G}) \subseteq S \text{ and } M_\mathcal{G} \vdash (h_\mathcal{G}, v_\mathcal{G}, \sigma) \Downarrow \sigma' \text{ then } \exists h, v.\ (h_\mathcal{G}, v_\mathcal{G}) \backsim (h, v) \text{ and } M \vdash (h, v, \sigma) \Downarrow \sigma'$$

In other words, a set of interleavings S is safe if for every interleaved history of $h_\mathcal{G}$ whose interleavings are a subset of S we can prove that $h_\mathcal{G}$ leads to the same final state as *some* history h of M where h simulates $h_\mathcal{G}$. This definition essentially lifts the second correctness criterion of Definition 3.16 to a fragmented monitor.

Inferring Safe Interleavings. We now turn our attention to the problem of *inferring* safe interleavings. Given a monitor M and its FDG $\mathcal{G} = (V, E)$, our goal is to find a set $\mathcal{I} \subseteq V \times E$ such that all interleavings in $\mathcal{I}$ are safe. However, a key challenge is that the space of all safe interleavings is exponential (i.e., the power set of $V \times E$), so, even if we had a procedure for checking whether some set $\mathcal{I}$ is safe, enumerating all candidates would be computationally intractable.

To overcome this challenge, we introduce the notion of *strong safety* that allows us to build $\mathcal{I}$ iteratively. In particular, note that if S_1 and S_2 are both safe interleaving sets according to Definition 4.10, it may *not* be the case that $S_1 \cup S_2$ is also a safe interleaving. However, to build $\mathcal{I}$ incrementally, we need a notion of safe interleaving that is closed under union. For this purpose, we introduce a notion of *strong safety* for a single interleaving (v, e). Since strongly safe interleavings enjoy the property of being closed under union, this notion lends itself to a computationally feasible technique for computing safe interleaving sets. In the remainder of this section, we define strong safety and present our static analysis for computing safe interleaving sets. Towards this goal, we first introduce the notions of *left* and *right commutativity* for our context:

Definition 4.11. **(Left/Right Commutativity).** Given fragments v and v', we say that v *left commutes* with v', denoted *LeftCommute*(v, v'), iff, whenever $M_\mathcal{G} \vdash ((v', t')(v, t), \nu, \sigma) \Downarrow \sigma'$ holds, so does $M_\mathcal{G} \vdash ((v, t)(v', t'), \nu, \sigma) \Downarrow \sigma'$. Conversely, v *right commutes* with v', denoted *RightCommute*(v, v'), iff $M_\mathcal{G} \vdash ((v, t)(v', t'), \nu, \sigma) \Downarrow \sigma'$ implies $M_\mathcal{G} \vdash ((v', t')(v, t), \nu, \sigma) \Downarrow \sigma'$.

In other words, a fragment v left commutes with v' if, whenever v executes just after v', the resulting state is the same as if v had executed just before v'. For example, f_4 (i.e. `count++`) in Figure 1a left-commutes with f_5 since increasing `count` right after `waituntil(count > 0)` is equivalent to increasing `count` just before `waituntil(count >0)`. That is, assuming that `waituntil(count>0)` was not blocked before executing `count++`, then it will still not be blocked *after* executing `count++`. However, f_4 does not left-commute with f_1: when `count` equals `queue.length - 1`, incrementing `count` just after `waituntil (count < queue.length)` is *not* equivalent to incrementing `queue.length` before the `waituntil` statement. That is, if `waituntil(count < queue.length)` did not block before executing `count++`, we cannot guarantee that it also does not block after executing `count++`.

Next, we use this notion of left and right commutativity to define strong safety:

Definition 4.12. **(Strong safety).** Let $\mathcal{G} = (V, E)$ be an FDG for monitor M, and let E^* denote the reflexive transitive closure of E. We say that an interleaving (v, e), where $e = (v_s, v_t)$, is *strongly safe* if the following conditions are satisfied:

(1) $\forall v^-.(v^-, v_s) \in E^* \implies \mathit{LeftCommute}(v, v^-)$
(2) $\forall v^+.(v_t, v^+) \in E^* \implies \mathit{RightCommute}(v, v^+)$

That is, an interleaving (v, e) is said to be *strongly safe* if we can prove that fragment v left commutes with *every* possible predecessor of v_s and that it right commutes with *every* possible successor of v_t. To see why these conditions imply safety, recall that a set of interleavings S is safe if, for any history $h_{\mathcal{G}}$ whose interleavings are a subset of S, we can find some (sequential) history of the original monitor that simulates $h_{\mathcal{G}}$. Assuming S contains only strongly safe interleavings, we can create such a sequential history by "removing" interleavings one at a time from $h_{\mathcal{G}}$. For instance, let $\chi = (v, (v_s, v_t)) \in S$ be an interleaving that occurs in $h_{\mathcal{G}}$. Since χ is strongly safe, we can always obtain an equivalent history $h'_{\mathcal{G}}$ that has strictly less interleavings than $h_{\mathcal{G}}$ by commuting v past either every successor of v_t or every predecessor of v_s that appears in $h_{\mathcal{G}}$.

Example 4.13. For the monitor from Figure 1a, we can show that every interleaving (v, e) where v belongs to method `take` and edge e belongs to method `put` (and vice versa) is strongly safe. However, none of the interleavings where v and e belong to the same method are strongly safe. Finally, because both of the interleavings of the history $h_{\mathcal{G}}$ from Eq. 1 are strongly safe, we can derive a sequential history that simulates history $h_{\mathcal{G}}$ by swapping (take_1, t') with (put_4, t).

We now state a key theorem that underlies the correctness of our approach:

Theorem 4.14. *Let $\mathcal{G}$ be an FDG and let $\chi_1, \ldots, \chi_n$ be* strongly safe *interleavings. Then, $\{\chi_1, \ldots, \chi_n\}$ satisfies Definition 4.10 (i.e., is a safe interleaving set for $\mathcal{G}$).*

Proof. Can be found in Appendix E. □

Static analysis algorithm. Finally, we conclude this section by presenting our static analysis algorithm (shown in Figure 7) for computing a set $\mathcal{I}$ of safe interleavings. At a high level, this algorithm identifies which (v, e) pairs are strongly safe and then returns their union, which by Theorem 4.14, corresponds to a safe interleaving set. To check whether an interleaving (v, e) (for $e = (v_s, v_t)$) is strongly safe, we must check if v left commutes with each predecessor of v_s and right commutes with each successor of v_t. As shown in the LeftCommute procedure, we reduce the verification of left commutativity to the problem of verifying a Hoare triple. In particular, given fragments v, v', we generate a code snippet $S_L; S_R$ where (1) S_L is an alpha-renamed version of $v'; v$ with `waituntil`'s replaced by `assume` statements, and (2) S_R is an alpha-renamed version of $v; v'$ with `waituntil`'s replaced by `assert` statements. Note that we turn `waituntil`'s in S_L into `assumes` because the definition of left commutativity assumes that $v'; v$ has terminated. On the

procedure FindSafeInterleavings($\mathcal{G}$)
  input: An FDG representation $\mathcal{G} = (V, E)$ of monitor M
  output: A set $\mathcal{I}$ of all safe interleavings
  $\mathcal{I} \leftarrow \emptyset$
  for $v \in V,\ e = (v_s, v_t) \in E$ **do**
    $V_s^* \leftarrow \{\ v' \mid (v', v_s) \in E^*\ \}$ ▷ All predecessor vertices that reach v_s.
    $V_t^* \leftarrow \{\ v' \mid (v_t, v') \in E^*\ \}$ ▷ All successor vertices of v_t.
    if $\left(\forall v_s^* \in V_s^*.\ \textsc{LeftCommute}(v, v_s^*)\right) \wedge \left(\forall v_t^* \in V_t^*.\ \textsc{LeftCommute}(v_t^*, v)\right)$ **then**
      $\mathcal{I} \leftarrow \mathcal{I} \cup \{(v, e)\}$
  return $\mathcal{I}$

function LeftCommute(v, v')
  input: Two fragments v, v'
  output: true iff v left commutes with v'
  $X \leftarrow \{x \mid x$ is a variable in v or $v'\}$.
  $X_L \leftarrow \{x_l$ fresh name $\mid x \in X\}$ $\quad X_R \leftarrow \{x_r$ fresh name $\mid x \in X\}$
  $S_L \leftarrow (v'; v)[\texttt{assume}/\texttt{waituntil}, X_L/X]$ $\quad S_R \leftarrow (v; v')[\texttt{assert}/\texttt{waituntil}, X_R/X]$
  return Verify($\{X_L = X_R\}\ S_L; S_R\ \{X_L = X_R\}$)

Fig. 7. Algorithm to find all safe interleavings.

other hand, we need to *show* that S_R does *not* block; thus, we assert that the predicates in the `waituntil` statement evaluate to true under the assumption that they also evaluate to true in S_L. Finally, in addition to showing that `waituntil`'s are not blocked, we also need to establish that the monitor state is the same in S_L and S_R. Thus, the Hoare triple we construct checks that the values of variables are the same at the end, assuming that they are the same in the beginning. Note that the implementation of right commutativity is the same with v and v' swapped; thus, RightCommute(v, v') can be checked by directly calling LeftCommute(v', v).

4.4 MaxSAT Encoding

In this section, we describe our MaxSAT encoding which is formalized as inference rules in Figure 8. Recall that the encoding procedure takes as input (a) an FDG representation of the monitor, (b) the results of the static analysis, and (c) an upper bound on the maximum number of locks, and it produces a set of hard constraints $\mathcal{H}$ and a set of soft constraints $\mathcal{S}$. In the remainder of this section, we describe the inference rules in Figure 8 for generating these constraints in more detail.

Variables. Our MaxSAT encoding uses two types of variables. First, we introduce variables of the form $h_{v_i}^{l_j}$ indicating that fragment v_i needs to hold lock l_j. Thus, given an FDG with n vertices and an upper bound $\mathcal{N}$ on the number of locks, our encoding contains $n \times \mathcal{N}$ such variables. The second type of variable used in our encoding is of the form a_{fld} indicating that *fld* should be implemented using an atomic type.

Mutex encoding. Given a set of fragments F and an upper bound $\mathcal{N}$ on the number of locks, we often need to enforce that all fragments in F share at least one of the $\mathcal{N}$ possible locks. We write

$$\textsc{Race-1}\quad \frac{IsFrag(v_1)\quad IsFrag(v_2)\quad Races = \mathcal{R}(v_1,v_2)\quad Races \subseteq \mathcal{F}\quad Races = \{\,\texttt{this.f}\,\}}{Mutex(\{v_1,v_2\},\mathcal{N}) \vee a_f \in \mathcal{H}}$$

$$\textsc{Race-2}\quad \frac{IsFrag(v_1)\quad IsFrag(v_2)\quad Races = \mathcal{R}(v_1,v_2)\quad Races \neq \emptyset\quad (|Races| > 1 \vee Races \not\subseteq \mathcal{F})}{Mutex(\{v_1,v_2\},\mathcal{N}) \in \mathcal{H}}$$

$$\textsc{I-Leave}\quad \frac{IsFrag(v)\quad IsEdge(e)\quad e = (v_s,v_t)\quad \neg SafeInterleaving(v,e)}{Mutex(\{v,v_s,v_t\},\mathcal{N}) \in \mathcal{H}}$$

$$\textsc{Wait}\quad \frac{p \in Preds(M)\quad F = \{\, f \mid IsFrag(f),\ f \equiv \texttt{waituntil(p)}\}}{Mutex(F,\mathcal{N}) \in \mathcal{H}\quad \bigwedge_{i=1}^{\mathcal{N}} \bigwedge_{v_1,v_2 \in F} \left(h_{v_1}^{l_i} \leftrightarrow h_{v_2}^{l_i}\right) \in \mathcal{H}}$$

$$\textsc{L-Order}\quad \frac{IsEdge(e)}{LockOrder(e,\mathcal{N}) \in \mathcal{H}}$$

$$\textsc{Min-Lock}\quad \frac{m \in Methods(M)\quad MF = \{\, v \mid IsFrag(v),\ Method(v) = m\,\}}{\bigcup_{i=1}^{\mathcal{N}} \{\, \bigwedge_{f \in MF} \neg h_f^{l_i}\} \subseteq \mathcal{S}}$$

$$\textsc{Min-Atom}\quad \frac{\texttt{this.fld} \in \mathcal{F}}{\neg a_{fld} \in \mathcal{S}}$$

$$\textsc{Max-Par}\quad \frac{IsFrag(v_1)\quad IsFrag(v_2)\quad \mathcal{R}(v_1,v_2) = \emptyset}{\neg Mutex(\{v_1,v_2\},\mathcal{N}) \in \mathcal{S}}$$

$$\textsc{Aux-Defs}\quad Mutex(F,\mathcal{N}) = \bigvee_{i=1}^{\mathcal{N}} \bigwedge_{f \in F} h_f^{l_i}\quad LockOrder((v_s,v_t),\mathcal{N}) = \bigwedge_{1 \le \ell < u \le \mathcal{N}} \neg\left(h_{v_s}^{u} \wedge h_{v_t}^{u} \wedge \neg h_{v_s}^{\ell} \wedge h_{v_t}^{\ell}\right)$$

Fig. 8. Inference rules for MaxSatEncoding$(M,\mathcal{G},\mathcal{S},\mathcal{N})$ procedure. $\mathcal{G} = (V,E)$ is an FDG of monitor M, $\mathcal{S} = (\mathcal{F},\mathcal{R},\mathcal{I})$ are the results of the static analysis, and $\mathcal{N}$ is an upper bound on the number of locks. Predicate $IsFrag(v)$ is true if $v \in V$, $IsEdge(e)$ if $e \in E$, and $SafeInterleaving(v,e)$ if $(v,e) \in \mathcal{I}$. Relations $Methods(M)$ and $Preds(M)$ return all methods of monitor M and all predicates that appear as an argument of a `waituntil` statement in M respectively.

$Mutex(F,\mathcal{N})$ to denote this requirement. In particular, as shown at the bottom of Figure 8, this is defined as $Mutex(F,\mathcal{N}) = \bigvee_{i=1}^{\mathcal{N}} \bigwedge_{f \in F} h_f^{l_i}$.

Hard constraints. Next, we describe the hard constraints generated by our MaxSAT encoding. These hard constraints $\mathcal{H}$ correspond to correctness requirements on the synthesized protocol and include (1) data race freedom, (2) correct signaling and deadlock freedom and (3) atomicity. Specifically, the first two rules in Figure 8 deal with data race freedom, the next rule deals with atomicity, and the last two rules deal with deadlock freedom and correct signaling.

Race-1. The first rule, labeled Race-1, deals with data race freedom of two fragments that have a data race on a *single* monitor field `f`. The premises of this rule stipulate that v_1, v_2 are fragments that race *only* on field f which can be converted to atomic (i.e., $\texttt{this.f} \in \mathcal{F}$). In this case, we prevent data races by either (1) enforcing that v_1, v_2 share a lock (the *Mutex* constraint) or (2) ensuring that field f is converted to an atomic field.

Race-2. The next Race-2 rule prevents data races between fragments where the data race cannot be resolved by making one of the fields atomic. In particular, given two fragments v_1, v_2 that have a data race, this rule simply enforces that they share a common lock via the *Mutex* function.

I-Leave. The next rule generates constraints to ensure that monitor operations appear to take place atomically. In particular, if the static analysis cannot prove (v, e) to be a strongly safe interleaving (recall Definition 4.12), then we need to ensure that a thread cannot execute v when some other thread is executing e. To do so, we ensure that v, v_s, v_t all share a common lock by generating a *Mutex* hard constraint for these three fragments.

L-Order. The next rule, labeled L-Order, ensures that the resulting synchronization protocol is deadlock-free. Specifically, for every edge $e = (v_s, v_t)$ in the input FDG, this rule generates a hard constraint, via $LockOrder(e, \mathcal{N})$ (defined at the bottom of Figure 8), that ensures that every lock acquisition respects the total order on locks. In particular, for every pair of locks l, u such that $l \prec u$, $LockOrder(e, \mathcal{N})$ prevents the synchronization protocol from violating the global order on locks. Recall that a protocol violates this global order if it acquires the "smaller" lock l between v_s and v_t while both v_s and v_t hold lock u. Thus, the hard constraint generated by $LockOrder(e, \mathcal{N})$ prevents this from happening.

Example 4.15. Assuming $\mathcal{N} = 2$, this rule generates $\neg \left(h^{l_2}_{v_s} \wedge h^{l_2}_{v_t} \wedge \neg h^{l_1}_{v_s} \wedge h^{l_1}_{v_t} \right)$ for edge (v_s, v_t).

Wait. The last hard constraint rule, called Wait, is used for associating a single lock with each condition variable. In particular, since all fragments of the form `waituntil`(p) must hold the same set of locks, this rule generates two hard constraints for every `waituntil` predicate p of the input monitor: (1) a mutex constraint for all `waituntil`(p) fragments and (2) a constraint that enforces that all `waituntil`(p) fragments must share *all* common locks.

Soft Constraints. As discussed earlier, our goal is to generate a synchronization protocol that is not only correct-by-construction but one that also results in efficient code. Hence, as a proxy metric for efficiency, we want to (1) minimize the number of locks and atomic fields that are introduced, and (2) maximize the number of fragments that can run in parallel. The remaining three rules in Figure 8 introduce soft constraints to encode this optimization objective.

Min-Lock. The rule labeled Min-Lock is used for minimizing the number of locks. However, instead of simply minimizing the total number of locks used by the protocol, the soft constraints generated by this rule minimize the number of locks used *per method*. Even though this is not equivalent to minimizing the number of locks used by the entire protocol, we have found this approach to synthesize protocols with a more even distribution of locks among the monitor methods. In practice, such protocols are more desirable because they avoid scenarios where a subset of the methods incur a higher synchronization cost than others. Specifically, this rule generates a soft constraint for every lock $l \in \{l_1 ... l_{\mathcal{N}}\}$ and every method m of M and asserts that none of the fragments in m hold lock l.

Min-Atom. The Min-Atom rule generates soft constrains to minimize the number of fields that are made atomic by asserting that a_{fld} is assigned to false.

Max-Par. The last rule called Max-Par generates soft constraints to maximize parallelism. Specifically, for every pair of fragments (v, v') that do not have data races, we add a soft constraint stating that v and v' *do not share* any locks.

We conclude this Section with a theorem that states the correctness of our MaxSAT encoding.

Theorem 4.16. *Let m be a model of the generated MaxSAT instance and $(\mathcal{L}, \mathcal{A}, \mathcal{P})$ be the synchronization protocol constructed as follows:*

$$\mathcal{L} = \left\{ v \mapsto \left\{ l \mid m[h^l_v] \right\} \right\} \quad \mathcal{A} = \left\{ \texttt{fld} \mid m[a_{fld}] \right\} \quad \mathcal{P} = \left\{ p \mapsto l_i \mid IsWait(v, p), i = min(\{ j \mid m[h^{l_j}_v] \}) \right\}$$

where, $IsWait(v, p)$ is true if v is a `waituntil` *statement on p. Then, $(\mathcal{L}, \mathcal{A}, \mathcal{P})$ is a correct synchronization protocol.*

Proof. Can be found in Appendix E. □

5 IMPLEMENTATION

We have implemented our approach in a tool called Cortado that emits explicit-synchronization monitors in Java. Cortado is based on the Soot program analysis infrastructure [Vallée-Rai et al. 1999] and the Z3 SMT solver [De Moura and Bjørner 2008]. In particular, we use Soot to perform various kinds of static analyses needed by our method (e.g., pointer analysis) and to translate the input monitor to an explicit-synchronization monitor in Java. Furthermore, we leverage Z3 for solving MaxSAT instances and discharging the validity queries that arise when checking commutativity between fragments. In the remainder of this section, we discuss several design choices and optimizations that were not discussed previously.

Weights of soft constraints. As expected, the quality of the synthesized protocol depends on the model returned by the MaxSAT solver. In practice, we have observed certain types of soft constraints to be more important than others for efficiency. Thus, our implementation assigns different weights for different classes of soft constraints. For instance, because it is always preferable to use an atomic field instead of a lock, Cortado assigns a higher weight to soft constraints generated by rule Min-Atom from Figure 8 than the ones generated by rule Min-Lock.[3]

Constructing FDGs. As mentioned in Section 4, Cortado uses a heuristic to partition the input CFG into fragments. The goal of this heuristic is to maximize parallelization opportunities while ensuring that the partition results in a valid FDG according to Definition 4.2. Our heuristic places every loop in its own fragment (to make sure that the FDG is well-formed) and, for code outside loops, Cortado creates a new fragment whenever it detects an update to monitor state (i.e., `this.fld = *`). In practice, we found this heuristic to achieve a good balance between the number of parallelization opportunities and the size of the resulting FDG.[4]

Static analysis optimization. Our approach uses an off-the-shelf pointer analysis to detect which pairs of FDG fragments do not have a data race (and, so, can run in parallel). However, such an approach, based on pointer analysis alone, often leads to imprecision. For example, Soot's pointer analysis cannot prove that fragments 2 and 6 in Figure 1a do not contain any races, as it does not reason about individual array elements. Therefore, in order to increase the precision of the static analysis, Cortado implements an SMT-based static analysis on top of Soot's built-in pointer analysis and generates appropriate verification conditions (similar to the ones generated by Gurfinkel et al. [2015]) to prove that memory accesses of two fragments are disjoint.

6 EVALUATION

We evaluated Cortado's ability to generate fine-grained locking protocols by performing a set of experiments that are designed to answer the following research questions:

RQ1 How does the code generated by Cortado compare against explicit-synchronization monitors written by experts?

RQ2 How does the technique implemented in Cortado compare against other compile-time state-of-the-art approaches targeting implicit-synchronization monitors?

[3] An ablation study that demonstrates the need for adjusting the weights of soft constraints can be found in Appendix A.3.
[4] An ablation study that justifies the design of this heuristic can be found in Appendix A.2.

RQ3 How does the static analysis for inferring safe interleavings impact the quality of the code generated by Cortado?

RQ4 How long does Cortado take to synthesize code and how complex are the resulting protocols?

To answer these research questions, we conducted experiments on ten explicit-synchronization monitors from popular open source repositories. Aside from Cortado, we consider two additional baselines, described below, that aid us in answering our second and third research questions.

Benchmarks. The benchmarks used in our evaluation are collected from popular open source GitHub repositories. We wrote a crawler (based on GitHub's REST API [GitHub 2022]) to automatically identify candidate explicit-synchronization monitors implemented in Java by searching for keywords like `lock`, `unlock`, `await`, etc. We then manually inspected class files returned by the crawler in decreasing order of GitHub popularity (stars and forks) and identified self-contained monitor-style classes that encapsulate shared state accessed by multiple threads. We included such a monitor in our benchmarks only if it satisfies the following conditions: (1) the class has a well-defined API for client threads and (2) it contains *parallelization opportunities that can be realized via fine-grained locking*.[5] We manually isolated the shared state and monitor methods of the class file to obtain a standalone explicit-synchronization monitor and then manually translated it to an equivalent implicit monitor. To convert a benchmark to an implicit monitor, we simply removed all synchronization code (i.e., locking and signaling operations) and introduced appropriate `waituntil` statements. In total, we collected 10 monitors from popular repositories such as Spring Framework (a Java-based framework for creating enterprise applications), Java JDK, Apache Spark (an analytics engine for large-scale data processing), etc.[6]

Baselines. As mentioned above, our evaluation uses two additional baselines in order to answer RQ2 and RQ3. To compare against other compile-time techniques (RQ2), we evaluate Expresso [Ferles et al. 2018], a tool that addresses the same problem as this paper but generates an explicit signal monitor using a *single global lock* shared by all monitor methods. To evaluate the importance of our static analysis (RQ3), we created an ablated version of Cortado, called Ablated, which uses a very coarse analysis to infer safe interleavings. This ablated version considers $(v, (v_s, v_t))$ a safe interleaving only if v does not have any data races with any predecessor (resp. successor) of v_s (resp. v_t). This is a sufficient condition for strong safety, but it only requires checking data races rather than discharging a set of Hoare triples.

Evaluating performance. Following prior work [Ferles et al. 2018; Hung and Garg 2013], we evaluate the performance of each monitor implementation by performing *saturation tests* [Cherem et al. 2008] wherein threads perform monitor operations without doing any additional work. We collect our performance measurements using the Java Microbenchmark Harness (JMH) [Shipilev et al. 2021]. All measurements are conducted on a 112-way (56-core × 2 SMT) Intel Xeon CPU W-3275 2.50GHz with 256 GB of memory using JDK 1.8.0_272. In this section, we present results for each benchmark for up to 128 threads, chosen as an arbitrary stopping point past the total number of hyper-threads. Results for up to 256 threads can be found in Appendix A.

[5] If a monitor does not contain parallelization opportunities, our technique generates code equivalent to that synthesized by Ferles et al. [2018]. Since the goal of our evaluation is to evaluate Cortado's ability to generate fine-grained locking protocols, we did not include benchmarks from prior work [Ferles et al. 2018; Hung and Garg 2013] that do not contain such parallelization opportunities.

[6] All benchmarks are publicly available here: https://github.com/utopia-group/cortado

6.1 Performance Results

Figure 9 plots the average time taken per monitor method invocation (i.e., milliseconds/operation) against the number of threads. In what follows, we analyze the plots in more detail and present several conclusions drawn from these results. Because our benchmarks only contain monitors where fine-grained locking is beneficial, we emphasize that our conclusions only apply to such monitors.

Comparison against hand-written implementations (RQ1). For every benchmark, the explicit synchronization monitor generated by Cortado performs *better* than the expert-written implementation as the number of threads increases. In particular, Cortado-synthesized code performs on average 3.7×[7] and up to 39.1× times faster than the original code.

Comparison against Expresso (RQ2). Cortado-generated explicit monitors perform better than Expresso explicit monitors generated from the same implicit specification on all benchmarks. Cortado-synthesized code outperforms Expresso by 4.0× on average (and up to 48.7×).

Comparison against Ablated (RQ3). Finally, we analyze how Cortado compares to its simplified version, Ablated, which does not use the results of the safe interleavings analysis from Section 4.3. In five cases, the code generated by Ablated is equivalent to the code generated by Expresso and therefore worse than Cortado. In two other cases (PausableThreadPoolExecutor and ProgressTracker), Ablated generates code different from both Expresso and Cortado. For PausableThreadPoolExecutor, the code generated by Ablated is slower than that of Expresso because many of the operations it parallelizes are very cheap, so the overhead of extra locks outweighs their benefit. On the other hand, our static analysis detects several safe interleavings which enable Cortado to synthesize a protocol with cheaper synchronization operations. Finally, for the remaining three cases, the code generated by Ablated matches the one generated by Cortado. This ablation study demonstrates that the safe interleaving analysis from Section 4.3 helps extract additional concurrency on five of our benchmarks.

6.2 Synthesis Time & Protocol Complexity

To evaluate the cost and complexity of synthesizing code with Cortado (RQ4), Table 1 summarizes its running time and presents some statistics about the synthesized protocols. For each benchmark, we report the running time for the various phases of the tool: pointer analysis with Soot, signal placement with Expresso, and synthesis with Cortado. We also report the number of locks and atomic fields in the synthesized protocol.

Table 1 shows that Cortado terminates in under one minute for all but two benchmarks. For these two outliers, the synthesis time is dominated by Expresso's monitor invariant inference, which is necessary for signal placement. Overall, Cortado is able to extract better performance than Expresso alone with only a small additional compile-time cost.

The last three columns in Table 1 provide statistics about the synthesized explicit monitors. Most monitors benefit from Cortado's ability to introduce atomic fields, which reduces the overhead of operations on monitor state that would otherwise require a lock. The lines of code (LOC) results show that Cortado synthesizes explicit monitors that are on average 1.7× larger than their implicit specifications.

[7] In order to handle outliers such as in JobWrapper, for all reported aggregate speedups (max, mean, etc.) we throw out data points with a z-score greater than two.

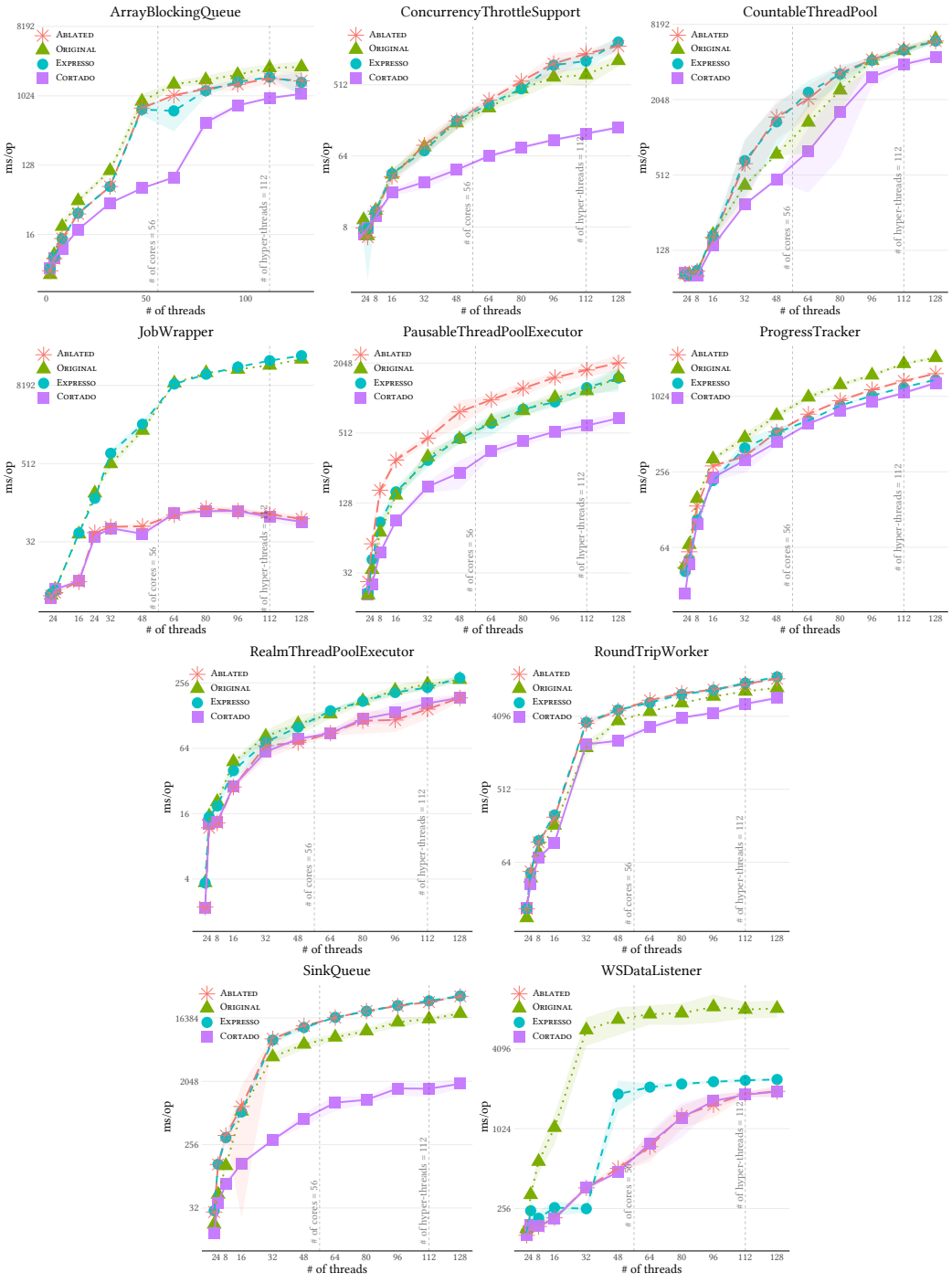

Fig. 9. Performance Results For All Tools. The y-axis is in log scale and time measurements are in milliseconds. The shadowed regions surrounding each line present 99.9% confidence interval of each measurement.

		Synthesis Time (secs)				Synthesized Protocol		
Benchmark	LOC	Soot	Expresso	Cortado	Total	#Lock/#Op	#Atomic/#Op	LOC
ArrayBlockingQueue	287	16.7	1897.7	372.7	2302.5	2 / 18	1 / 25	514
ConcurrencyThrottleSupport	33	17.0	1.4	0.2	18.7	1 / 1	1 / 4	68
CountableThreadPool	54	20.4	0.8	0.2	21.5	1 / 1	1 / 5	85
JobWrapper	33	17.0	0.6	0.1	17.7	1 / 1	1 / 3	63
PausableThreadPoolExecutor	79	20.7	0.8	0.9	22.5	3 / 4	2 / 9	122
ProgressTracker	65	0.7	6.8	0.2	8.0	2 / 7	1 / 4	119
RealmThreadPoolExecutor	34	18.4	0.3	0.1	18.7	1 / 1	1 / 3	61
RoundTripWorker	62	16.7	3.1	0.4	20.4	2 / 4	1 / 4	103
SinkQueue	75	15.7	981.4	34.6	1031.8	2 / 4	1 / 8	131
WSDataListener	158	18.3	5.2	1.0	25.1	4 / 11	0 / 0	222

Table 1. Synthesis time for each phase of Cortado and summaries of the synthesized protocols. LOC is lines of code. Soot indicates pointer analysis time, Expresso is the time for monitor invariant generation and signal placement, and Cortado shows the additional time on top of Soot and Expresso.

7 RELATED WORK

Monitor abstractions. The notion of monitors as an organizing abstraction for concurrent programming originates with Hoare [1974] and Hansen [1973]. Monitors offer the same synchronization facilities as semaphores—the ability to coordinate multiple threads and enforce mutual exclusion—but encapsulate the state protected by those facilities and automate mutual exclusion when entering and exiting the monitor's operations. Lampson and Redell [1980] further extended the monitor abstraction in the Mesa programming language to handle spurious wake ups.

These early monitor abstractions are *explicit-signal* monitors in the taxonomy of Buhr et al. [1995] because they require the programmer to explicitly insert condition variables and signalling operations to coordinate threads within the monitor. This requirement places both a safety and a liveness burden on the programmer: they must place signals correctly to preserve invariants about the monitor's state, but must also insert enough signals to avoid deadlock. An alternative is to use an *implicit-signal* (or *automatic-signal*) monitor, in which signals are inserted automatically by the compiler, language runtime, or operating system. Hoare [1971] proposed the notion of *conditional critical regions* (CCRs), which allow for monitor operations to block until a *guard* predicate over the monitor state is satisfied by some other thread. A CCR implementation would automatically block and signal threads in a fashion consistent with this guard semantics.

Implicit-signal monitors simplify concurrent programming, but come at a steep performance cost—Buhr et al. [1995] estimate that implicit-signal monitors are 10–50× slower than explicit ones. More recent work has tried to lower the cost of implicit-signal monitors. AutoSynch [Hung and Garg 2013] uses a combination of compile-time instrumentation and run-time evaluation to efficiently compute which threads should be woken when monitor state changes. This approach lowers the cost of implicit monitors to be close to, or sometimes better than, explicit ones. Expresso [Ferles et al. 2018] takes a different approach, using compile-time static analysis to *synthesize* an explicit-signal monitor equivalent to an implicit-signal version given as input. In this way, Expresso is able to erase the dynamic cost of implicit-signal monitors, and in most cases is comparable to hand-written explicit monitors. However, Expresso uses a single lock for the entire monitor and does not allow concurrent execution of threads within the monitor even when safe. Our work expands on this direction by using a richer static analysis to infer additional concurrency opportunities and uses MaxSAT to synthesize a safe and efficient locking protocol. Hence, our key contribution is to synthesize an explicit monitor that *appears* to match the semantics of the implicit one, but runs

monitor operations concurrently when possible and efficient. As we show in the evaluation, our proposed approach can often make the synthesized monitor *faster* than a hand-written equivalent.

Automatic synchronization. An appealing approach to lower the difficulty of concurrent programming is to deploy program analysis and synthesis techniques for automation. The common abstraction for much of this work is for the programmer to annotate *atomic sections* that should be executed atomically. Emmi et al. [2007] present a technique for lock allocation to an annotated program. They reduce the problem to integer linear programming and deploy the resulting tool on large-scale C and Java programs. Other approaches [Halpert et al. 2007; Hicks et al. 2006; McCloskey et al. 2006], on the other hand, take a purely static analysis route and attempt to maximize parallelism based solely on the results of the analysis. Cherem et al. [2008] present an alternative technique that uses runtime support to enable finer-grained concurrency. Compared to these efforts, Cortado applies to the more limited domain of monitors, but in exchange for this limitation is able to reason about conditional signalling and can allow atomic sections to run concurrently so long as the illusion of atomicity is maintained.

Other approaches start from a sequential program and automatically generate an equivalent concurrent program. The closest work to ours in this space is that of Golan-Gueta et al. [2011] which generates concurrent data structures given their sequential implementation. Compared to our method, their approach is applicable only to data structures that satisfy certain shape properties and all synthesized programs adhere to the same locking protocol, whereas Cortado generates a synchronization protocol specialized to the input monitor.

Concurrency verification. Cortado reasons about concurrent program executions by building on work in concurrent program analysis and verification. Our notion of left- and right-commutativity (Definition 4.11) comes from Lipton's work on reduction as a concurrency proof technique [Lipton 1975]. Reduction translates interleaved program executions to simpler, equivalent sequential executions by exploiting the commutativity properties of individual program steps. We use the same idea but in reverse: starting with a sequential history (Definition 3.7), we use a static analysis of commutativity to determine how to safely introduce interleavings into that history, and use that information to determine how to assign locks to program fragments.

8 CONCLUSION

We presented a technique for synthesizing fine-grained synchronization protocols for implicitly synchronized monitors. Our approach first employs a novel static analysis to identify safe interleavings opportunities between code fragments and uses the results of this analysis to generate a MaxSAT encoding whose solution can be used to synthesize an efficient and correct-by-construction explicit-synchronization monitor. We have implemented our method in a tool called Cortado and evaluated its effectiveness eight monitors collected from popular open source applications. The results of our experimental evaluation demonstrate that Cortado is able to generate non-trivial synchronization protocols that are 3.7× times faster than the original implementation on average (and up to 39.1× times for some outliers).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Noga Alon. 1986. Covering graphs by the minimum number of equivalence relations. *Combinatorica* 6, 3 (Sep 1986), 201–206. https://doi.org/10.1007/BF02579381

Andrew D Birrell. 1989. *An introduction to programming with threads.* Digital Systems Research Center.

Peter A. Buhr, Michael Fortier, and Michael H. Coffin. 1995. Monitor Classification. *ACM Comput. Surv.* 27, 1 (1995), 63–107.

Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. 2008. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Tucson, AZ, USA, 304–315.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. 2007. Lock allocation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007.* 291–296.

Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. 2018. Symbolic Reasoning for Automatic Signal Placement. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018).* Association for Computing Machinery, New York, NY, USA, 120–134. https://doi.org/10.1145/3192366.3192395

GitHub. 2022. GitHub REST API. https://docs.github.com/en/rest

Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G Ramalingam, Mooly Sagiv, and Eran Yahav. 2011. Automatic fine-grain locking using shape properties. *ACM SIGPLAN Notices* 46, 10 (2011), 225–242.

Arie Gurfinkel, Temesghen Kahsai, and Jorge A Navas. 2015. SeaHorn: A framework for verifying C programs (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 447–450.

Richard L Halpert, Christopher JF Pickett, and Clark Verbrugge. 2007. Component-based lock allocation. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007).* IEEE, 353–364.

Per Brinch Hansen. 1973. *Operating System Principles.* Prentice-Hall.

Michael Hicks, Jeffrey S Foster, and Polyvios Pratikakis. 2006. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.*

C. A. R. Hoare. 1971. Towards a theory of parallel programming. In *Operating Systems Techniques, Proceedings of a Seminar at Queen's University, Belfast.* Belfast, Northern Ireland.

C. A. R. Hoare. 1974. Monitors: An Operating System Structuring Concept. *Commun. ACM* 17, 10 (1974), 549–557.

Wei-Lun Hung and Vijay K. Garg. 2013. AutoSynch: an automatic-signal monitor based on predicate tagging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI.* Seattle, WA, USA, 253–262.

Butler W. Lampson and David D. Redell. 1980. Experience with Processes and Monitors in Mesa. *Commun. ACM* 23, 2 (1980), 105–117.

William Landi and Barbara G. Ryder. 1992. A Safe Approximate Algorithm for Interprocedural Aliasing. *SIGPLAN Not.* 27, 7 (July 1992), 235–248. https://doi.org/10.1145/143103.143137

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 346–358.

TS Michael and Thomas Quint. 2006. Sphericity, cubicity, and edge clique covers of graphs. *Discrete Applied Mathematics* 154, 8 (2006), 1309–1313.

Aleksey Shipilev, Sergey Kuksenko, Astrand Astrand, Staffan Freiberg, and Henrik Loef. 2021. OpenJDK: jmh. http://openjdk.java.net/projects/code-tools/jmh/

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research.* IBM Press, 13.

A ADDITIONAL EXPERIMENTAL EVALUATIONS

This section presents additional experimental data for the evaluation of Section 6 as well as ablations related to several design decisions in the implementation of Cortado. In particular, Section A.1 presents additional data points for the evaluation presented in Section 6, Section A.2 presents an ablation study that justifies the design of our heuristic for constructing an FDG, and Section A.3 presents an ablation study that demonstrates the need for adjusting the weights of soft constraints in our MaxSAT encoding.

A.1 Additional Data Points per Benchmark

In this Section we present additional data points for all the experiments presented in Section 6. As mentioned in Section 6, we chose 128 threads as a stopping point past the number of total hyper-threads in the machine used in our evaluation. In this Section, we provide data points up to 256 threads.

Figure 10 presents the results for all benchmarks in our evaluation up to 256 threads. As demonstrated by Figure 10, the general trend for all benchmarks is the same as the date presented in Section 6. Note that for benchmarks where the code generated by Cortado exhibits similar run-time performance with the other three implementation we consider in our evaluation (e.g., RoundTripWorker), context-switches seem to dominate the running time as the number of threads increases.

A.2 Ablation Study for FDG Construction Heuristic

This section presents an ablation study for the design of our FDG construction heuristic described in Section 5. To justify the decisions behind the design of our heuristic, we implemented two additional versions of Cortado that differ in the way they construct the FDG. In particular, we implemented a version that creates finer-grained FDGs than Cortado and one that creates coarser-grained ones. The finer-grained version, named Stmt-Ablation, puts every non-composite statement outside of a loop in its own fragment. Statements inside a loop are grouped together in the same fragment since FDGs are acyclic. The coarser version of our tool, named CCR-Ablation, simply puts each CCR in its own fragment.

Figure 11 presents the results of this ablation study. As demonstrated by the results, there are several cases where Cortado performs better than at least one of its two modified versions. The cases where all three versions perform similarly are benchmarks where the synthesized synchronization protocol mainly exploits data-level parallelism among different CCRs in the monitor, which our tool can exploit given any FDG. Overall, this study demonstrates the need for a customized heuristic for constructing an FDG suitable for maximizing parallelism of implicit synchronisation monitors.

A.3 Ablation Study for MaxSAT Soft Constraint Weights

Finally, this Section presents an ablation of the soft constraints weights in our MaxSAT encoding. As mentioned in Section 5, Cortado assigns different weights to different classes of soft constraints because some of them are more important for synthesizing the optimal synchronization protocol. To demonstrate this, we have created a modified version of our tool, named Weight-Ablation, that assigns the same weight to all soft constraints.

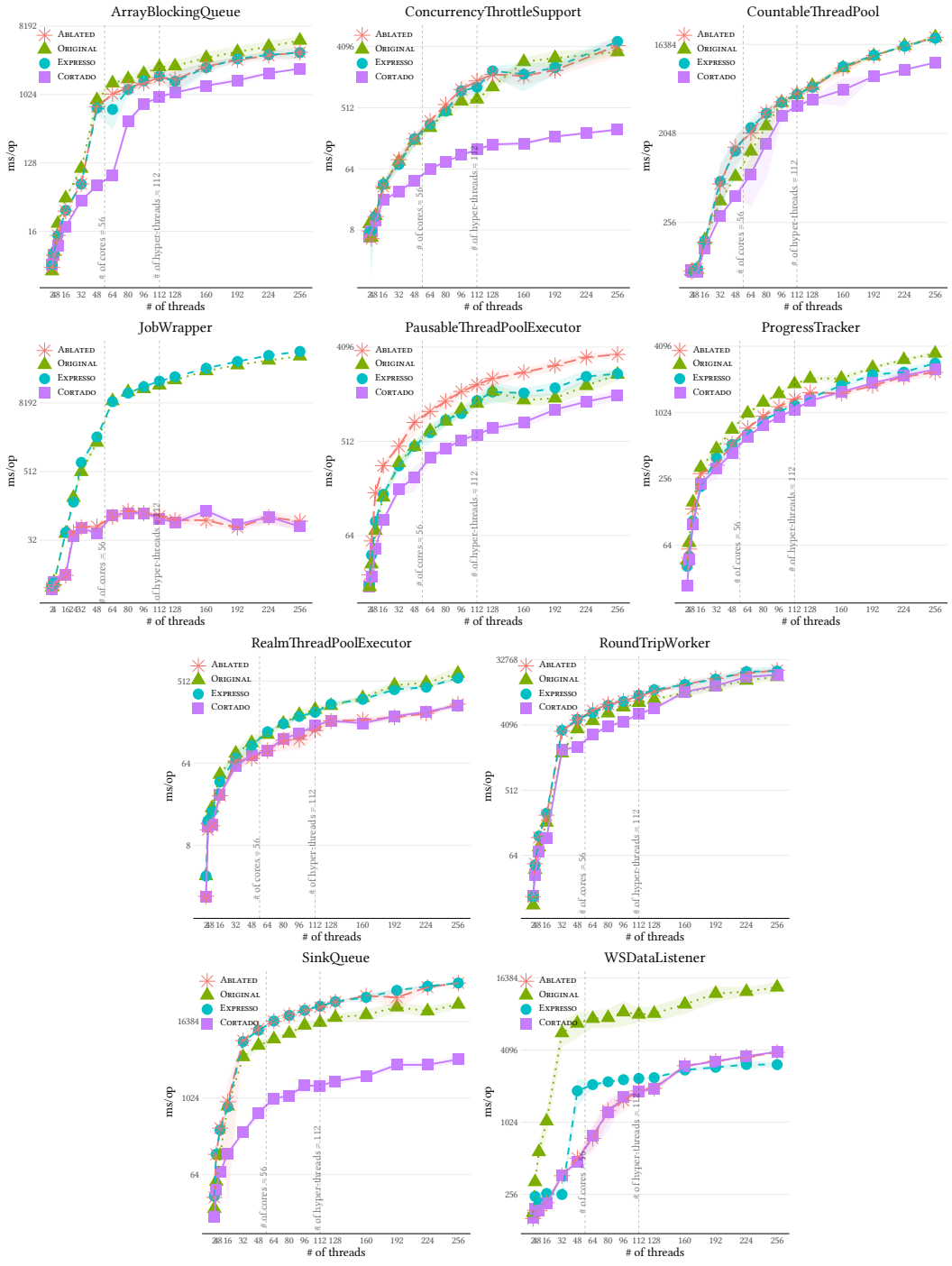

Fig. 10. Performance results for all tools up to 256 threads. The y-axis is in log scale and time measurements are in milliseconds. The shadowed regions surrounding each line present 99.9% confidence interval of each measurement.

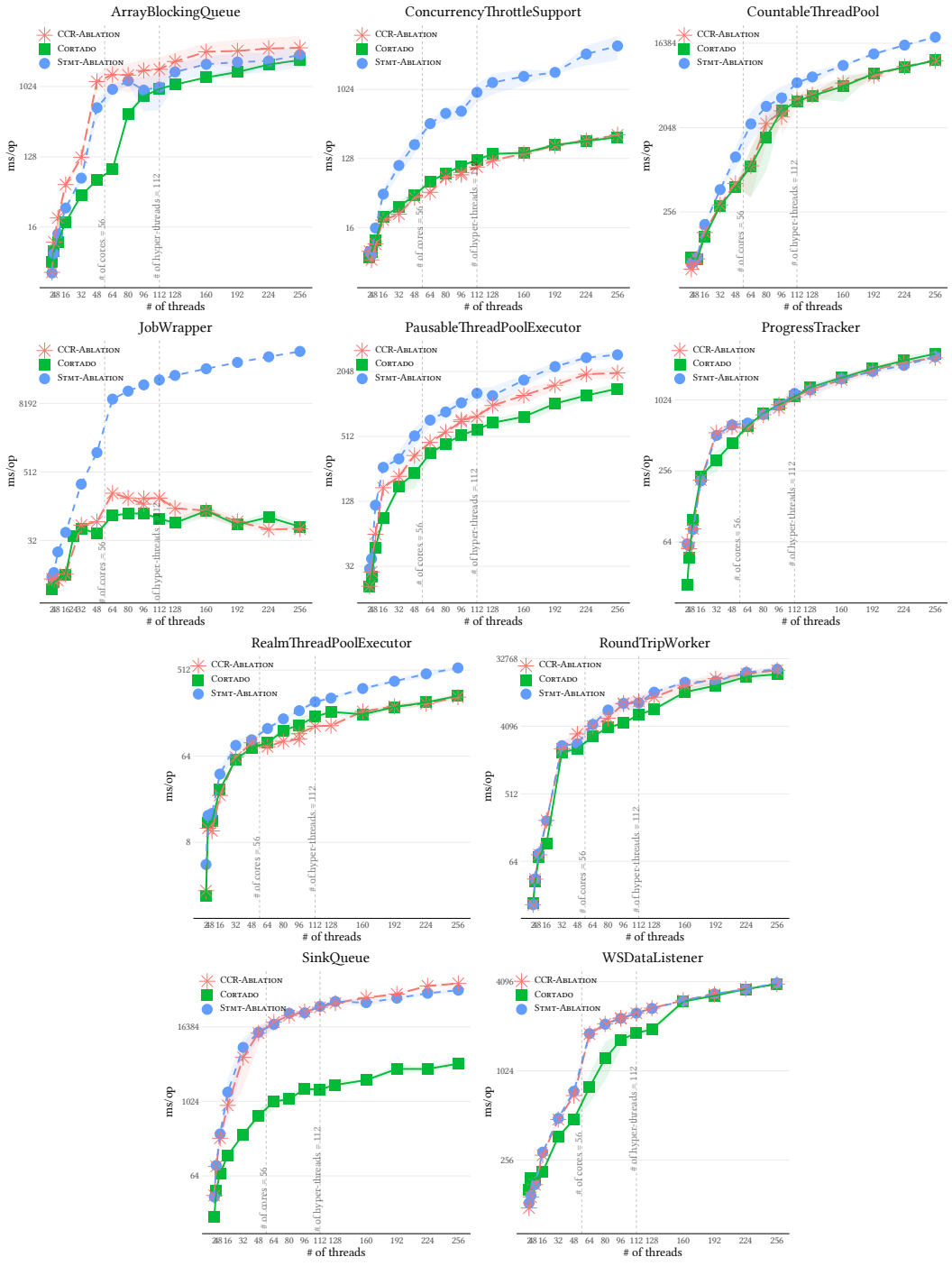

Fig. 11. Performance results for the FDG ablation study. The y-axis is in log scale and time measurements are in milliseconds. The shadowed regions surrounding each line present 99.9% confidence interval of each measurement.

The results of this ablation are presented in Figure 13. As this figure demonstrates, there are several cases where the ablated version of the tool performs significantly worse than CORTADO. To give a concrete example of why this is the case, consider the monitor of Figure 12 that contains two methods both of which modify field x. Because the body of method `bar` can be interleaved between the `waituntil` statement and the increment statement of method `foo`, the optimal synchronization protocol would convert field x to an atomic integer and introduce a lock that would only be held in method `foo`. However, an equivalent protocol would be to simply protect both `foo` and `bar` with the same global lock. So, if all constraints have an equal weight, CORTADO could generate both of these protocols, since they would have the same optimum objective value. As mentioned in Section 5, CORTADO's MaxSAT encoding prefers assignments where a race between two fragments (like the one on field x) are resolved via an atomic field rather than a lock. This forces CORTADO to generate the optimal solution for the monitor of Figure 13. The adjusted weights for other classes of soft constraints try to steer CORTADO to better performing synchronization protocols in a similar way.

```
class M {
  int x = 0;
  void foo() {
    waituntil(x < 10);
    x++;
  }

  void bar() {
      x--;
  }
}
```

Fig. 12. A simple implicit monitor.

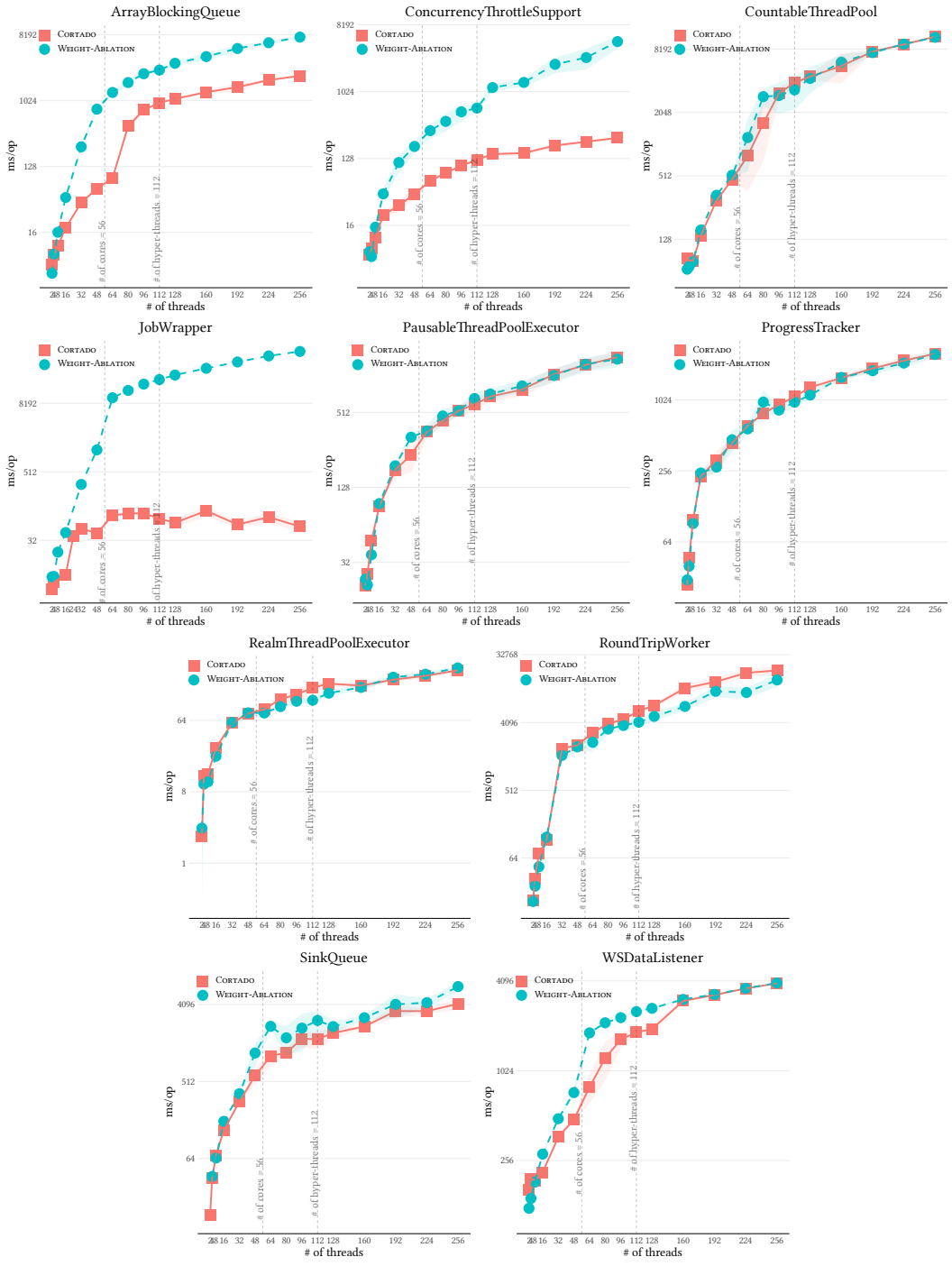

Fig. 13. Performance results for the soft constraints weights ablation. The y-axis is in log scale and time measurements are in milliseconds. The shadowed regions surrounding each line present 99.9% confidence interval of each measurement.

B PROOF OF NP-COMPLETENESS

To aid the reader, we restate Theorem 4.4.

Theorem B.1. **(NP-Completeness)** *Let $\mathcal{G} = (V, E)$ be the FDG representation of a monitor M and let $\Pi \subseteq V \times V$ be a set of fragment pairs that can safely run in parallel. Then, deciding whether there exists a synchronization protocol with at most k locks and atomic fields that allows all pairs in Π to run in parallel is an NP-Complete problem.*

Proof of Theorem 4.4. We prove the theorem by reduction to the edge clique cover problem [Michael and Quint 2006]. Let $G = (V, E)$ be an undirected graph. For each $v \in V$, let $E(v)$ be the set of edges incident to v.

Define monitor M as follows: for each edge $e \in E$ define a new field f_e of the monitor, initially set to zero. For each vertex $v \in V$, define a CCR $inc_v()$ which increments each f_e for $e \in E(V)$, has a guard of $\top$, and returns nothing.

Let G_M be the control-flow graph of monitor M. Note that there is one $waituntil(\top)$ statement for each $inc_v()$ and $|E(v)| = degree(v)$ increment statements in $inc_v()$, so there are $|V| + 2|E|$ total nodes in G_M. Define $\{G_{1,M}, \ldots, G_{|V|+2|E|,M}\}$ to be the partition of G_M into singletons. Then, let $\mathcal{G}_M = (V_M, E_M)$ to be the fragment dependency graph obtained from this partition, and let $\Pi \subseteq V_M \times V_M$ be the set of fragment pairs that can safely run in parallel.

We write $frag_{v,e}$ for the fragment which increments f_e in method $inc_v()$, and $waituntil_v$ for the $waituntil(\top)$ statement at the beginning of $inc_v()$. Observe that

$$\begin{aligned}\Pi =& \left\{(f_1, f_2) \mid \exists v \in V \text{ such that } f_1 = waituntil_v, \text{ or } f_2 = waituntil_v\right\} \\ &\cup \left\{(frag_{v_1,e_1}, frag_{v_2,e_2}) \mid e_1 \neq e_2\right\}.\end{aligned} \tag{2}$$

Note that any synchronization protocol which implements f_e for some $e = (u, v) \in E$ as an atomic variable is equivalent to one which wraps a unique lock around each $frag_{u,e}$ and $frag_{v,e}$. Therefore, we only need to consider synchronization protocols which use only locks.

Suppose we are given a synchronization protocol which allows all pairs in Π to run in parallel and uses exactly k locks, $\{\ell_1, \ldots, \ell_k\}$, for some $k \in \mathbb{N}$. Define the vertices holding each lock to be

$$C_i = \left\{v \in V \mid \exists e \in E(v) \text{ such that } frag_{v,e} \text{ holds lock } \ell_i\right\}. \tag{3}$$

We claim that $\{C_1, \ldots, C_k\}$ is a clique edge cover of G. First, observe that every edge $e = (u, v) \in E$ corresponds to two fragments in $\mathcal{G}_M$: $frag_{u,e}$ and $frag_{v,e}$. Since these fragments must not run in parallel (due to a data race), they must share some lock. Let ℓ_i be that lock. By the definition of C_i in Equation 3, $u \in C_i$ and $v \in C_i$. Therefore, every edge appears in C_i for some $1 \leq i \leq k$. Second, suppose that $u \neq v \in V$ are both contained in C_i for some i. By Equation 3, there must be some $e_u \in E(u)$ and $e_v \in E(v)$ such that both $frag_{v,e_v}$ and $frag_{u,e_u}$ hold lock i. Since the two fragments share a lock, we know $(frag_{v,e_v}, frag_{u,e_u}) \notin \Pi$. Therefore, $e_v = e_u$ (by Equation 2). Hence, there is an edge $e_u = e_v = (u, v)$ in E. Consequently, any two distinct vertices in C_i are incident, so C_i is a clique.

A symmetric argument shows how to construct a synchronization protocol using exactly k locks from any edge clique-cover of G which has k cliques.

We have shown that, given an arbitrary graph G, in polynomial time we may compute a monitor M such that M has a synchronization protocol using at most k locks and atomic variables if and only if G has an edge clique-cover with at most k cliques.

□

$$(1)\quad \frac{e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad \neg\textsc{Synch}(s) \quad (s,v),\sigma \Downarrow \sigma' \quad \mathcal{T}' = \textsc{UpdateState}(\mathcal{T},t)}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(2)\quad \frac{\begin{array}{c}e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{l.lock()} \\ \textsc{LockHeld}(\mathcal{T},t,\sigma[l]) \quad \mathcal{T}' = \textsc{BlockThreadOnLock}(\mathcal{T},t,\sigma[l]) \quad \textsc{NoDeadLocks}(\mathcal{T}')\end{array}}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(3)\quad \frac{\begin{array}{c}e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{l.lock()} \\ \neg\textsc{LockHeld}(\mathcal{T},t,\sigma[l]) \quad \mathcal{T}' = \textsc{AcqLock}(\mathcal{T},t,\sigma[l])\end{array}}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(4)\quad \frac{e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{l.unlock()} \quad \mathcal{T}' = \textsc{RelLock}(\mathcal{T},t,\sigma[l])}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(5)\quad \frac{e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{c.await()} \quad \mathcal{T}' = \textsc{BlockOnCVar}(\mathcal{T},t,\sigma[c])}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(6)\quad \frac{e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{c.signal()} \quad \mathcal{T}' = \textsc{SigCVar}(\mathcal{T},\sigma[c])}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(7)\quad \frac{e = (s,t) \quad \textsc{CheckNotif}(\mathcal{T},t) \quad s = \texttt{c.signalAll()} \quad \mathcal{T}' = \textsc{BCastCVar}(\mathcal{T},\sigma[c])}{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}$$

$$(8)\quad \frac{(\sigma,e,v,\mathcal{T}) \Rightarrow (\sigma',\epsilon,\epsilon,\mathcal{T}')}{(\sigma,e :: h,v :: v',\mathcal{T}) \Rightarrow (\sigma',h,v',\mathcal{T}')}$$

Fig. 14. Semantics for our target language 2b. Here, his a history, v a list of arguments for every element in h, and $\mathcal{T}$ a tuple of sets and mappings that keep track of all pending signaling and locking operations. Methods and predicates that appear in small caps are defined the text.

C TARGET LANGUAGE OPERATIONAL SEMANTICS

This section presents the semantics of our target language presented in Figure 2b. As mentioned in Section 3, given an explicit monitor M_t, initial state σ, and monitor history h_e with argument mapping v_e, the operational semantics of M_t is defined using a judgment $M_t \vdash (h_e, v_e, \sigma) \downarrow \sigma'$ indicating that the new state is σ' after executing h_e on initial state σ. The semantics of such a monitor are implemented using the inference rules of Figure 14 that use judgements of the form $(\sigma, h_e, v_e, \mathcal{T}) \Rightarrow (\sigma', h'_e, v'_e, \mathcal{T}')$. Here, $\mathcal{T}$ and $\mathcal{T}'$ are tuples of the form $(\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L)$ and their role is to keep track all signaling and locking operations of the history. Specifically, each element of the tuple is defined below:

- $\mathcal{B}_S \subseteq T \times CVar$: a set of thread-condition variable pairs, $(t,c) \in \mathcal{B}_S$ means that thread t is blocked on condition variable c.
- $\mathcal{N}_S \subseteq T \times CVar$: a set of thread-condition variable pairs, $(t,c) \in \mathcal{N}_S$ means that thread t has been notified on condition c.
- $\mathcal{H}_L \subseteq T \times L$: a set of thread-lock pairs, $(t,l) \in \mathcal{H}_L$ means that thread t holds lock l.
- $\mathcal{B}_L \subseteq T \times L$: a set of thread-lock pairs, $(t,l) \in \mathcal{B}_L$ means that thread t is blocked waiting to acquire lock l.

1: **procedure** UpdateState($\mathcal{T}, t$)
2: **input:** $\mathcal{T} = (\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L)$
3: **input:** t, thread executing a non-synchronization statement.
4: **output:** updated mappings.
5: **if** $(t, c) \in \mathcal{B}_S$ **then**
6: $\mathcal{B}'_S \leftarrow \mathcal{B}_S \setminus \{(t, c)\}$ $\mathcal{N}'_S \leftarrow \mathcal{N}_S \setminus \{(t, c)\}$
7: **if** $(t, l) \in \mathcal{B}_L$ **then**
8: $\mathcal{H}'_L \leftarrow \mathcal{H}_L \cup \{(t, l)\}$
9: $\mathcal{N}'_L \leftarrow \mathcal{N}_L \setminus \{(t, l)\}$
10: $\mathcal{B}'_L \leftarrow \mathcal{B}_L \setminus \{(t, l)\}$
11: **return** $(\mathcal{B}'_S, \mathcal{N}'_S, \mathcal{H}'_L, \mathcal{B}_L, \mathcal{N}'_L)$

Fig. 15. Procedure UpdateState

- $\mathcal{N}_L \subseteq T \times L$: a set of thread-lock pairs, $(t, l) \in \mathcal{N}_L$ means that thread t can acquire a previously held lock l

We say that $M_t \vdash (h_e, v_e, \sigma) \downarrow \sigma'$ if and only if $(\sigma, h_e, v_e, \mathcal{T}) \Rightarrow^* (\sigma', \epsilon, \epsilon, \mathcal{T}')$, where $\Rightarrow^*$ is the reflexive transitive closure of relation $\Rightarrow$. In other words, a h_e is a valid explicit history according to our operational semantics only if the rules of Figure 14 can "consume" the entire history. If none of the rules of Figure 14 apply to a history, then we consider the computation of relation $\Rightarrow$ stuck and thus the history is not valid.

On a high level, the rules of our operational semantics iterate over all statements of input history h and updates the sets inside $\mathcal{T}$ accordingly. Because during the execution of a h a thread t might perform a blocking operation (e.g., call `l.lock` on a state where `l` is being held), the rules require every statement to be executed in a state where a thread is not blocked. To ensure this, every rule in Figure 14 requires predicate $M_t \vdash (h_e, v_e, \sigma) \downarrow \sigma'$, defined below, to hold for the executing thread t.

$$\textsc{CheckNotif}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t) = (t, c) \in \mathcal{B}_S \leftrightarrow (t, c) \in \mathcal{N}_S \ \wedge \ (t, l) \in \mathcal{B}_L \leftrightarrow (t, l) \in \mathcal{N}_L$$

Essentially, $\textsc{CheckNotif}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t)$ requires every thread t that was previously blocked by some operation ($(t, \_) \in \mathcal{B}_S$ or $(t, l) \in \mathcal{B}_L$) to be first notified ($(t, \_) \in \mathcal{N}_S$ or $(t, l) \in \mathcal{N}_L$) in order to execute a statement.

In what follows, we explain each of the rules of Figure 14 in more detail.

Rule (1). This rule applies for all statements that are not a synchronization statement (i.e., lock or signal operation). Because the operational semantics for non-synchronization statements are well-studied, we assume the existence of an oracle $\Downarrow$ that give a statement s and its argument v, it returns the resulting monitor state σ'. Furthermore, because thread t could be blocked before executing statement s, this rule uses procedure UpdateState (defined in Figure 15) to update the sets inside $\mathcal{T}$ accordingly. Specifically, if thread t was blocked in some condition variable c, then procedure UpdateState removes pair (t, c) from both $\mathcal{B}_S$ and $\mathcal{N}_S$ (recall that if t was blocked then it is guaranteed to be notified). Similarly, if thread t was blocked on some lock l, then procedure

UpdateState add the pair (t, l) to $\mathcal{H}_L$ (i.e., now t holds lock l) and removes it from $\mathcal{N}_L$ and $\mathcal{B}_L$ (same as in the condition variable case).

Rule (2). This rule applies to all statements where a thread t attempts to acquire lock 1 that is currently held by another thread. In order to determine whether a lock is held by another thread, this rule makes use of predicate LockHeld defined as follows:

$$\textsc{LockHeld}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t, l) = l \in \mathcal{H}_L[t']. \ t \neq t'$$

Then, the rule marks thread t as blocked on lock l by using the following procedure that updates map $\mathcal{B}_L$ by adding pair (t, l):

$$\textsc{BlockThreadOnLock}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t, l) = (\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}'_L, \mathcal{N}_L)$$
$$\text{where } \mathcal{B}'_L = \mathcal{B}_L \setminus \{(t, l)\}$$

Finally, the rule requires that the new attempt to acquire lock l does not introduce any deadlocks by invoking oracle NoDeadLocks. This oracle detects any cycles in the lock acquisition by examining maps $\mathcal{B}_L$ and $\mathcal{H}_L$.

Rule (3). Conversely, the third rule applies to all cases where thread thread t attempts to acquire a lock not currently held by some other thread. In this case, the rule simply adds pair (t, l) in map $\mathcal{H}_L$ as follows:

$$\textsc{AcqLock}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t, l) = (\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}'_L, \mathcal{B}_L, \mathcal{N}_L)$$
$$\text{where } \mathcal{H}'_L = \mathcal{H}_L \cup \{(t, l)\}$$

Rule (4). This rule is triggered when a thread t releases lock l. The rule performs the following two updates to maps $\mathcal{H}_L$ and $\mathcal{N}_L$:

$$\textsc{RelLock}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t, l) = (\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}'_L, \mathcal{B}_L, \mathcal{N}'_L)$$
$$\text{where } \mathcal{H}'_L = \mathcal{H}_L \setminus \{(t, l)\}, \ \mathcal{N}'_L = \mathcal{N}_L \cup \{(t', l)\} \text{ s.t. } (t', l) \in \mathcal{B}_L$$

Specifically, it removes pair (t, l) from $\mathcal{H}_L$ and notifies some thread t' currently blocked on lock l.

Rule (5). This rule applies when a thread t calls method `await` on a condition variable c. The rule simply adds pair (t, c) in set $\mathcal{B}_S$.

$$\textsc{BlockOnCVar}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), t, c) = (\mathcal{B}'_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L)$$
$$\text{where } \mathcal{B}'_S = \mathcal{B}_S \setminus \{(t, c)\}$$

Rules (6) and (7). These two rules are used when a thread signals or broadcasts a condition variable c. They simply update set $\mathcal{N}_S$ as follows:

$$\textsc{SigCVar}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), c) = (\mathcal{B}_S, \mathcal{N}'_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L)$$
$$\text{where } \mathcal{N}'_S = \mathcal{N}_S \cup \{(t, c)\} \text{ s.t. } (t, c) \in \mathcal{B}_S$$

$$\textsc{BcastCVar}((\mathcal{B}_S, \mathcal{N}_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L), c) = (\mathcal{B}_S, \mathcal{N}'_S, \mathcal{H}_L, \mathcal{B}_L, \mathcal{N}_L)$$
$$\text{where } \mathcal{N}'_S = \mathcal{N}_S \cup \{(t, c) \mid (t, c) \in \mathcal{B}_S\}$$

Specifically, rule 6 adds *a single* thread t currently blocked on condition variable c in $\mathcal{N}_S$, whereas rule 7 adds *all* such threads in $\mathcal{N}_S$.

Rule (8). Finally, rule 8 recursively applies the procedure to the whole input history h.

D MONITOR INSTRUMENTATION.

In this section, we describe procedure Instrument which given an implicit-synchronization monitor M, it corresponding FGD $\mathcal{G} = (V, E)$, and a synchronization protocol $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{P})$, it instrument protocol $\mathcal{S}$ into M yielding an explicit-synchronization monitor M' equivalent to M. This is achieved by first introducing all the necessary synchronization fields (locks, condition variables, and atomic fields) in the input class and then instrumenting locking and signaling operations in all methods as follows:

- **Lock acquisition and release:** The synthesized code must ensure that all the locks in $\mathcal{L}(f)$ are held when executing fragment f. Thus, for every edge (f, f') in the FDG, we instrument the code to acquire locks $\mathcal{L}(f')\backslash\mathcal{L}(f)$ and release locks $\mathcal{L}(f)\backslash\mathcal{L}(f')$. Furthermore, as mentioned in Section 2, we acquire and release these locks according to a static total order to prevent deadlocks.
- **Blocking on predicates:** Our instrumentation must also convert every `waituntil` statement to a sequence of operations on locks and condition variables. Specifically, we instrument a `waituntil(p)` statement as follows:

  ```
  while(!p) {  ln.unlock(); ...; l2.unlock(); c.await(); l2.lock(); ...; ln.lock(); }
  ```

  where `c` is the condition variable associated with `p`; `l1, ... ln` are the locks associated with this fragment, and `l1` is the lock associated with condition variable `c`.
- **Signaling operations:** Finally, we instrument signaling operations introduced by PlaceSignals to acquire and release the appropriate locks. In particular, given a statement `signal(p,c)` (similarly for `broadcast(p,c)`), our instrumentation generates the following code:

  ```
  if (c)  { lp.lock(); cp.signal(); lp.release(); }
  ```

  where `cp` is the condition variable for predicate `p` and `lp` is the corresponding lock for `cp`.

Procedure Instrument is presented in Figure 16 in the form of inference rules that use the following two judgements:

- $v \vdash \Delta \rightsquigarrow \Delta'$, where v is a subset of the arguments of procedure Instrument (we overload operator $\rightsquigarrow$ depending on the arguments) and Δ is one of the following: the input monitor, a field, a method, a CCR or a statement.
- $\mathcal{L}, \mathcal{G} \vdash v \hookrightarrow v'$, where $\mathcal{L}$ is the lock map of the input synchronization protocol $\mathcal{S}$, FDG is the input $\mathcal{G}$, and v is a fragment in $\mathcal{G}$.

The meaning of each judgement is that whenever procedure Instrument is applied to an element that appears on the left-hand side of an arrow ($\rightsquigarrow$, $\hookrightarrow$) it generates the element on the right-hand side.

Overall Structure. The core logic of this procedure is to recursively iterate every element of the input monitor and use the inferred synchronization protocol in order to convert each element to an equivalent element of the target language. At a top level, the procedure begins by transforming every field and method of the input monitor. For every method, the procedure recursively visits every CCR using operator $\rightsquigarrow$. Then, for every CCR, it collects all its fragments and uses operator $\hookrightarrow$ to instrument all the lock operations dictated by the input protocol. In what follows, we give a brief description of every rule presented in Figure 16.

Mtr. This is the top-level rule called by procedure Instrument and performs the following tasks: 1. it introduces all the synchronization fields (locks and condition variables) needed by the synchronization protocol and initializes them accordingly and 2. it recursively calls itself for every field, and method of M.

Fld-1 & Fld-2. These two rules are used to translate fields of M, with the first one being applicable to fields that must be converted to atomic fields and the second one to fields that should remain the

same. Only the first rule alters the original field by converting to an atomic field with the same name as the original.

Method & CCR.. These two rules simply recursively apply operator $\rightsquigarrow$ to their constituent elements.

CCR-Statement. This rule is the one that splits each top-level CCR-Statement to a set of fragments that belong in the input FGG $\mathcal{G}$ and then recursively transforms each of the fragments. Note, because of the properties of FDG (Definitions 4.1 and 4.2), there is only one way to decompose each CCR to its constituents fragments.

Frag-Stmt. This rule applies to all statements s that are a fragment in the input $\mathcal{G}$. It first uses operator $\hookrightarrow$ to instrument all necessary lock operations and then uses a special oracle $\rightarrow_{\mathcal{A}}$ that converts all operations that involve a field converted to atomic to the equivalent `update` statement in the target language.[8]

Wait. Rule labeled Wait is a special case of the above rule because, by definition, every `waituntil` statement defines its own fragment in an FDG. Similar to the rule above, this rule also uses operator $\hookrightarrow$ to instrument the appropriate lock operation in the fragment but it additionally translates the `waituntil` statement into an equivalent statement in the target language that uses condition variables. As mentioned in Section 4, each `waituntil` statement is translated into a while loop that waits on the appropriate condition variable and properly releases and acquires all locks before and after the call to method `await`. Additionally, it acquires all locks needed by its successor statement v in the FDG and releases all locks held by it but not needed by v (similar to the logic described below).

Sig. This rule applies to all fragments that are a signalling directive of the monitor's intermediate representation.[9] In a similar manner as the rule for `waituntil` statements, this rule first uses operator $\hookrightarrow$ to instrument all lock operations needed to implement the synchronization protocol. Then, it consults the predicate map $\mathcal{P}$ of the synchronization protocol to acquire the appropriate lock and perform the signaling operation on the associated condition variable.

Instrumenting Fragments With Lock Operations. Finally, we describe operator $\hookrightarrow$ which given a fragment v, the lock map $\mathcal{L}$ of the input synchronization protocol $\mathcal{S}$, and the FDG $\mathcal{G}$, it instruments all the necessary lock operations. The logic of this operator is split between two groups of rules, described in more detail below:

- Rules for entry & exit fragments (i.e., fragments without predecessors and successors respectively), which are handled by rules Entry-Frag and Exit-Frag respectively. These rules simply lookup the entry or exit fragment in $\mathcal{L}$ and acquire or release the locks returned by the $\mathcal{L}$ accordingly.
- Rules for fragments with successors. Fragments that contain some successor in the graph are handled by rules Branch-Frag, Reg-Frag-1 and Reg-Frag-2. The logic for each of these rule is similar, i.e., for any successor fragment v_s of fragment v, the instrumentation releases all locks required by v but not by v_s ($\mathcal{L}[v] \setminus \mathcal{L}[v_s]$) and acquires all locks required by v_s but not v ($\mathcal{L}[v_s] \setminus \mathcal{L}[v]$). All these operation are operation in accordance to the global lock order to prevent deadlocks. Last, it is worth mentioning that the main difference of these three rules is how they instrument the edge between v and v_s. That is, if v ends with a `goto` statement

[8] Due to its simplicity, we omit a formal description of oracle $\rightarrow_{\mathcal{A}}$.
[9] For simplicity, we assume that every signaling operation defines its own fragment.

(conditional or not), then the instrumentation redirects the control-flow appropriately so all lock operations occur along edge (v, v_s).

Finally, we conclude with the following theorem that states the correctness of our instrumentation phase.

Theorem D.1. *Let $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{P})$ be a synchronization protocol inferred over FGD $\mathcal{G} = (V, E)$ of input monitor M and M' be the result of procedure Instrument for M. Then, the following three conditions hold:*

(1) For every fragment $v \in V$, $l_i \in \mathcal{L}[v]$ iff fragment v holds lock l_i in M'
(2) If $i < j$, then l_i is never acquired whenever l_j is held.
(3) Field $f \in \mathcal{A}$ iff all its occurrences in M have been replaced with an atomic operation in M'.

Proof. Proof can be find in Appendix E. □

E CORRECTNESS-RELATED PROOFS

This section contains all the proofs related to the correctness of our approach. Section E.1 presents the proof of Theorem 4.5, Section E.2 presents the proof of Theorem 4.14, Section E.3 presents the proof of Theorem 4.16, and Section E.4 presents the proof of Theorem D.1.

E.1 Proof of Theorem 4.5

Theorem 4.5 states the following:

(Correctness). We say that an explicit monitor M_t correctly implements an implicit monitor M_s, denoted as $M_s \sim M_t$, iff for all input states σ_s, σ_t s.t. $\sigma_s \equiv_{M_s} \sigma_t$, we have:

(1) $\forall h_i, v_i.\ M_s \vdash (h_i, v_i, \sigma_s) \Downarrow \sigma'_s \implies \left(M_t \vdash (\mathsf{Expand}_{M_t}(h_i, v_i, \sigma_s), \sigma_t) \downarrow \sigma'_t \ \wedge\ \sigma'_s \equiv_{M_s} \sigma'_t\right)$

(2) $\forall h_e, v_e.\ M_t \vdash (h_e, v_e, \sigma_t) \downarrow \sigma'_t \implies \left(\exists h_i, v_i.\ (h_e, v_e) \rightharpoonup (h_i, v_i) \wedge M_s \vdash (h_i, v_i, \sigma_s) \Downarrow \sigma'_s \wedge \sigma'_s \equiv_{M_s} \sigma'_t\right)$

Proof. For all proofs in this Section, we assume the correctness of procedure PlaceSingals (proved in previous work [Ferles et al. 2018]).

The proof of condition (1) above follows directly from Theorems 4.16, D.1, and the correctness of procedure PlaceSingals. The proof of condition (2) follows directly from Theorem 4.14, Theorem 4.16, Theorem D.1, and correctness of PlaceSignals. □

E.2 Proof of Theorem 4.14

In this section, we present the proof of Theorem 4.14 which we reiterate here for convenience.

Before presenting the actual proof, we first introduce some auxiliary notation, relations, and lemmas. First, given a history h, we define a predicate $h[\![(v_1, t_1)_{i1}, \ldots, (v_k, t_k)_{ik}]\!]$ that evaluates to true iff each event $(v_i, t_i)_i$ is i-th element in h and $i1 < \ldots < ik$. That is, this predicate encodes that these event occur in this particular order within h. Second, we define $Next(h, i, t)$ as follows: $min(\{j \mid j > i,\ h_{\mathcal{G}}[\![(\_, t)_j]\!]\})$. In other words, $Next(h, i, t)$ returns the first element in h after index i whose thread identifier is t. Additionally, we use $h[i]$ to denote the i-th element in h and $h[i : j],\ i < j$ to denote the "sub-history" of h between its i-th element (inclusive) and j-th element (exclusive). Finally, we extend the definition of a history projection to filter out elements that *do not* involve a thread , e.g., $\Pi(h, \neg t)$ filters out all events of h that involve thread t.

Next, using the notation above we define some relations that identify interleavings inside a history of a fragmented monitor $M_{\mathcal{G}}$.

Entry-Frag
$$\frac{\neg\exists v_p.\ (v_p, v) \in E \quad A = \mathcal{L}[v]}{\mathcal{L}, (F, E) \vdash v \hookrightarrow Acq(A); v}$$

Exit-Frag
$$\frac{\neg\exists v_s.\ (v, v_s) \in E \quad R = \mathcal{L}[v]}{\mathcal{L}, (F, E) \vdash v \hookrightarrow v; Rel(R)}$$

Branch-Frag
$$\frac{\begin{array}{c} Exit(v) \equiv \texttt{if (c) goto l} \quad (v, v_{s1}) \in E \quad (v, v_{s2}) \in E \quad v_{s1} \neq v_{s2} \quad v_{s2} \equiv \texttt{l: s} \\ (A_i, R_i) = (\mathcal{L}[v_{si}] \setminus \mathcal{L}[v], \mathcal{L}[v] \setminus \mathcal{L}[v_{si}]),\ i \in \{1, 2\} \quad v' = v[\texttt{l'/l}] \\ e_1 = (Rel(R_1); Acq(A_1); \texttt{goto l"}) \quad e_2 = (\texttt{l' :} Rel(R_2); Acq(A_2); \texttt{goto l}) \end{array}}{\mathcal{L}, (F, E) \vdash v \hookrightarrow v'; e_1; e_2; \texttt{l":\ skip}}$$

Reg-Frag-1
$$\frac{\begin{array}{c} Exit(v) \equiv \texttt{goto l} \quad (v, v_s) \in E \quad (A, R) = (\mathcal{L}[v_s] \setminus \mathcal{L}[v], \mathcal{L}[v] \setminus \mathcal{L}[v_s]) \\ v' = v[\texttt{l'/l}]; (\texttt{l' :} Rel(R); Acq(A); \texttt{goto l}) \end{array}}{\mathcal{L}, (F, E) \vdash v \hookrightarrow v'}$$

Reg-Frag-2
$$\frac{Exit(v) \not\equiv \texttt{goto l} \quad (v, v_s) \in E \quad (A, R) = (\mathcal{L}[v_s] \setminus \mathcal{L}[v], \mathcal{L}[v] \setminus \mathcal{L}[v_s])}{\mathcal{L}, (F, E) \vdash v \hookrightarrow v; Rel(R); Acq(A)}$$

Wait
$$\frac{\begin{array}{c} \mathcal{L}, (F, E) \vdash w \hookrightarrow w' \quad (\texttt{c}, \texttt{e}) = NewLabels() \quad l_p = \mathcal{P}[w] \\ L_w = \mathcal{L}[w] \quad rel = Rel(L_w \setminus \{l_p\}) \quad acq = Acq(L_w \setminus \{l_p\}) \\ (w, v) \in E \quad (A, R) = (\mathcal{L}[w] \setminus \mathcal{L}[v], \mathcal{L}[v] \setminus \mathcal{L}[w]) \quad succLocks \equiv Rel(R); Acq(A) \\ w'' = w'[(\texttt{c: if (p) goto e});(\texttt{rel;c}_p\texttt{.await();acq;goto c});(\texttt{e: succLocks})/w] \end{array}}{(\mathcal{L}, \mathcal{A}, \mathcal{P}), (F, E) \vdash w \equiv \texttt{waituntil}(p) \rightsquigarrow w''}$$

Frag-Stmt
$$\frac{IsFrag(s) \quad \mathcal{L}, \mathcal{G} \vdash s \hookrightarrow s' \quad s' \rightarrow_{\mathcal{A}} s''}{(\mathcal{L}, \mathcal{A}, \mathcal{P}), \mathcal{G} \vdash s \rightsquigarrow s''}$$

Sig
$$\frac{\begin{array}{c} SigOp(s) \quad \mathcal{L}, \mathcal{G} \vdash s \hookrightarrow s' \\ s'' = s'[ExplSig(s, \mathcal{P})/s] \end{array}}{(\mathcal{L}, \mathcal{A}, \mathcal{P}), \mathcal{G} \vdash s \rightsquigarrow s'}$$

CCR-Statement
$$\frac{\mathcal{G} \equiv (F, E) \quad s \equiv s_1; \ldots; s_n \quad s_i \in F \quad s_i \rightsquigarrow s_i'}{\mathcal{S}, \mathcal{G} \vdash s \rightsquigarrow s_1'; \ldots; s_n'}$$

Method
$$\frac{\mathcal{S}, \mathcal{G} \vdash c_i \rightsquigarrow c_i'}{\mathcal{S}, \mathcal{G} \vdash m(\vec{v})\{c_1 \ldots c_n\} \rightsquigarrow m(\vec{v})\{c_1' \ldots c_n'\}}$$

CCR
$$\frac{\begin{array}{c} \mathcal{S}, \mathcal{G} \vdash s \rightsquigarrow s' \\ \mathcal{S}, \mathcal{G} \vdash w \rightsquigarrow w' \end{array}}{\mathcal{S}, \mathcal{G} \vdash w; s \rightsquigarrow w'; s'}$$

Fld-1
$$\frac{\texttt{fld} \in \mathcal{A}}{\mathcal{A} \vdash \tau\ \texttt{fld} := e \rightsquigarrow \texttt{Atomic[}\tau\texttt{]\ fld} := e}$$

Fld-2
$$\frac{\texttt{fld} \notin \mathcal{A}}{\mathcal{A} \vdash \tau\ \texttt{fld} := e \rightsquigarrow \tau\ \texttt{fld} := e}$$

Mtr
$$\frac{\begin{array}{c} \mathcal{S} \equiv (\mathcal{L}, \mathcal{A}, \mathcal{P}) \quad \mathcal{G} \equiv (F, E) \quad l_i \triangleq \texttt{Lock}\ l_j\ \texttt{:= new Lock()}\ s.t.\ l_j \in Locks(\mathcal{L}) \\ cv_i \triangleq \texttt{CondVar cv}_p\ \texttt{:= } l_j\texttt{.newCV()}\ s.t.\ (p, l_j) \in \mathcal{P} \quad \mathcal{A} \vdash f_i \rightsquigarrow f_i' \quad \mathcal{S}, \mathcal{G} \vdash m_i \rightsquigarrow m_i' \end{array}}{\mathcal{S}, \mathcal{G} \vdash \texttt{mtr}\ M\ \{\ f_1 \ldots f_m\ m_1 \ldots m_n\ \} \rightsquigarrow \texttt{mtr}\ M\ \{\ l_1 \ldots l_k\ cv_1 \ldots cv_l\ f_1' \ldots f_m'\ m_1' \ldots m_n'\ \}}$$

Aux-Defs

$$Acq(L) \triangleq l_{i_1}\texttt{.lock();}\ldots\texttt{;}l_{i_k}\texttt{.lock()}\ s.t.\ l_{i_j} \in L, \forall j.\ 1 \leq j \leq k, i_j < i_{j+1}$$

$$Rel(L) \triangleq l_{i_k}\texttt{.unlock();}\ldots\texttt{;}l_{i_1}\texttt{.unlock()}\ s.t.\ l_{i_j} \in L, \forall j.\ 1 \leq j \leq k, i_j < i_{j+1}$$

$$ExplSig(s, \mathcal{P}) = \begin{cases} \texttt{if (c) \{ l.lock();c}_p\texttt{.signal();l.unlock();\}} & s \equiv signal(p, c), l \equiv \mathcal{P}[p] \\ \texttt{if (c) \{ l.lock();c}_p\texttt{.signalAll();l.unlock();\}} & s \equiv bcast(p, c), l \equiv \mathcal{P}[p] \end{cases}$$

Fig. 16. Procedure Instrument($\mathcal{S}, \mathcal{G}$), where $\mathcal{S}$ is a synchronization protocol for FDG $\mathcal{G}$.

Definition E.1. **(History Interleaving).** Given history $h_{fdg} = (V, E)$ and interleaving $\chi = (v, e = (v_s, v_t))$. We define the occurrences of χ as follows:

$$Interleavings(\chi, h_{\mathcal{G}}) = [(j, (i, k)) \mid h_{\mathcal{G}}[\![(v_s, t)_i, (v, t')_j, (v_t, t)_k]\!],\ t \neq t',\ k = Next(h_{\mathcal{G}}, i, t)]$$

Also, we write $\mathcal{X}(h_{\mathcal{G}})$ to be the set of all interleavings that occur in $h_{\mathcal{G}}$, i.e.

$$\mathcal{X}(h_{\mathcal{G}}) = \{\chi \mid X = Interleavings(\chi, h_{\mathcal{G}}), |X| > 0\}$$

Finally, we write $\mathcal{X}_{\#}(h_{\mathcal{G}})$ to denote the *number* of interleavings inside h. Formally:

$$\mathcal{X}_{\#}(h_{\mathcal{G}}) = \sum_{\chi \in V \times E} |Interleavings(\chi, h_{\mathcal{G}})|$$

Next, given a fragment v we assume the existence of two predicates, namely, *EntryFrag* and *ExitFrag*, that hold only if v is the entry fragment or the exit fragment of its CCR respectively. Based on these relations, we define the next relation that partitions a fragment history into CCR sub-histories.

CCR History Partition. Let $h_{\mathcal{G}}$ be a history of fragments in FDG $\mathcal{G}$. We define the CCR partition of $h_{\mathcal{G}}$ that returns a list of potentially overlapping sub-histories of $h_{\mathcal{G}}$ as follows:

$$CCRPart(h_{\mathcal{G}}) = \left[h_{\mathcal{G}}[i:j] \;\middle|\; \begin{array}{c} h_{\mathcal{G}}[i] = (v_{in}, t, \_),\ EntryFrag(v_{in}), \\ j = min(\{k \mid k > i,\ h_{\mathcal{G}}[k+1] = (v_{out}, t, \_), ExitFrag(v_{out})\}) \end{array} \right]$$

Let $P = CCRPart(h_{\mathcal{G}})$, then we use $P[i]$ to refer to the i-th sub-history in P. Note we assume that partitions returned by *CCRPart* are ordered according to the index of the first element in the sub-history. That is, if ccr_1 began its execution before ccr_2 in $h_{\mathcal{G}}$, then the partition of ccr_1 appears before the partition of ccr_2 in P. Furthermore, given a CCR partition $P[i]$, we write $Thread(P[i])$ to represent the thread of the first element in sub-history $P[i]$.

Removing Interleavings from Histories. Before we prove our main theorem, we define some transformations on interleaved histories that helps us remove interleavings.

First, given a CCR sub-history that is interleaved, we define its sequential history as follows:

Definition E.2. **Sequential CCR Sub-history** Let $h_{\mathcal{G}}$ be an interleaved history, $P = CCRPart(h_{\mathcal{G}})$, and $h'_{\mathcal{G}} = P[i]$ be an interleaved CCR partition. We define that the sequential history of $h'_{\mathcal{G}}$, denoted as $Seq_{|CCR}(h'_{\mathcal{G}})$ to be the following history: $\Pi(h'_{\mathcal{G}}, t)\Pi(h'_{\mathcal{G}}, \neg t)$, where $t = Thread(P[i])$. Given an argument mapping v for history $h'_{\mathcal{G}}$, we write $Seq(v)$ to denote the corresponding argument mapping for $Seq_{|CCR}(h'_{\mathcal{G}})$.

We now prove the following useful lemmas about sequential CCR sub-histories.

Lemma E.3. *Let $h_{\mathcal{G}}$ be an interleaved sub-history and $h'_{\mathcal{G}} = Seq_{|CCR}(h_{\mathcal{G}})$, then the following two things hold:*

(1) $\mathcal{X}(h'_{\mathcal{G}}) \subseteq \mathcal{X}(h_{\mathcal{G}})$

(2) $\mathcal{X}_{\#}(h'_{\mathcal{G}}) < \mathcal{X}_{\#}(h_{\mathcal{G}})$

Proof. Both of the properties logically follow from the construction of $h'_{\mathcal{G}}$. That is, a sequential CCR sub-history of the form $(v_1, t) \ldots (v_i, t), (v_{i+1}, t'), \ldots, (v_j, t'')$, where all elements before the i-th position are from thread t and all elements past that are from some thread t' s.t. $t \neq t'$. On the other hand, the original history $h_{\mathcal{G}}$ is of the form:

$$(v_1, t) \ldots (v_k, t)(v_{k+1}, t') \ldots (v_j, t)$$

Where $h_{\mathcal{G}}[1:k+1] = h'_{\mathcal{G}}[1:k+1]$ (i.e., $h_{\mathcal{G}}$ and $h'_{\mathcal{G}}$ have a common prefix). Therefore, since by its construction $h'_{\mathcal{G}}$ does not move the relative order of element is $h_{\mathcal{G}}$ that do not involve thread t,

if an interleaving $\chi \in \mathcal{X}(h'_{\mathcal{G}})$ then we also have $\chi \in \mathcal{X}(h_{\mathcal{G}})$. Conversely, any thread interleaving that involved thread t in $h_{\mathcal{G}}$ does not appear in $h_{\mathcal{G}}$ (by construction). Since by its definition the interleaved history $h_{\mathcal{G}}$ contains at least one interleaving that involves an edge executed by t, we can conclude that $\mathcal{X}_{\#}(h'_{\mathcal{G}}) < \mathcal{X}_{\#}(h_{\mathcal{G}})$. □

Lemma E.4. *Let $h_{\mathcal{G}}$ be an interleaved sub-history and $h'_{\mathcal{G}} = Seq_{|CCR}(h_{\mathcal{G}})$, then if $\mathcal{X}(h_{\mathcal{G}})$ is a set of* strongly *safe interleavings we have that:* $\forall \sigma, v. M_{\mathcal{G}} \vdash (h_{\mathcal{G}}, v, \sigma) \Downarrow \sigma' \Rightarrow M_{\mathcal{G}} \vdash (h'_{\mathcal{G}}, Seq(v), \sigma) \Downarrow \sigma'$

Proof. We prove this by induction on the number of distinct interleavings of history $h_{\mathcal{G}}$ ($\mathcal{X}_{\#} h_{\mathcal{G}}$).

Base Case: $\mathcal{X}_{\#}(h_{\mathcal{G}}) = 1$. If there is a single interleaving in $h_{\mathcal{G}}$, this implies that $h_{\mathcal{G}}$ is of the form:

$$(v_1, t) \dots (v_i, t') \dots (v_j, t)$$

where $t = Thread(h_{\mathcal{G}})$ and (v_i, t') is the only element in $h_{\mathcal{G}}$ not executed by t. Because, the interleaving of $h_{\mathcal{G}}$ is strongly safe, we have that fragment v_i executed by t', *right commutes with any possible successor of the edge it interleaves.* Also, by definition, $h'_{\mathcal{G}}$ is $(v_1, t) \dots (v_j, t)(v_i, t')$. Combining this two facts with lemma E.3, we can prove the theorem for our base case:

$$\forall \sigma, v. M_{\mathcal{G}} \vdash (h_{\mathcal{G}}, v, \sigma) \Downarrow \sigma' \rightarrow M_{\mathcal{G}} \vdash (h'_{\mathcal{G}}, Seq(v), \sigma) \Downarrow \sigma'$$

Inductive Step. In our inductive step, we assume that our lemma holds for $\mathcal{X}_{\#}(h_{\mathcal{G}}) = n$ and we are going to prove it for $n + 1$. The logic is similar to the base case, specifically, we get the right-most interleaved fragment in $h_{\mathcal{G}}$ and right-commute to the end of the history while obtaining a semantically equivalent history $h''_{\mathcal{G}}$. After that, we can apply our inductive hypothesis on $h''_{\mathcal{G}}$, which again proves our goal. □

Finally, we prove our main theorem, which we re-iterate below for convenience.

Theorem 4.14. Let $\mathcal{G}$ be an FDG and let $\chi_1, \dots, \chi_n$ be *strongly safe* interleavings. Then, $S = \{\chi_1, \dots, \chi_n\}$ is a safe interleaving set for $\mathcal{G}$.

Proof. By definition of safe set of interleavings, we have to prove the following for every interleaved history $h_{\mathcal{G}}$ of monitor $M_{\mathcal{G}}$

$$\text{If } \mathcal{X}(h_{\mathcal{G}}) \subseteq S \text{ and } M_{\mathcal{G}} \vdash (h_{\mathcal{G}}, v_{\mathcal{G}}, \sigma) \Downarrow \sigma' \text{ then } \exists h, v.\ (h_{\mathcal{G}}, v_{\mathcal{G}}) \backsim (h, v) \text{ and } M \vdash (h, v, \sigma) \Downarrow \sigma'$$

In order to prove that, we have to prove that for every interleaved history $h_{\mathcal{G}}$ that only allows interleavings in S and argument mapping $v_{\mathcal{G}}$ we can find a history of the original monitor with corresponding argument mapping s.t., $((h_{\mathcal{G}}, v_{\mathcal{G}}) \backsim (h, v)$. Which in turn means that we have to find a sequential history of $M_{\mathcal{G}}$ $h'_{\mathcal{G}}$ s.t.

$$(1)\ \forall t.\ \pi(h_{\mathcal{G}}, t) = \pi(h'_{\mathcal{G}}, t) \qquad \textit{and} \qquad (2)\ \mathsf{Expand}_{M_{\mathcal{G}}}(h, v, \sigma) = (h'_{\mathcal{G}}, v_{\mathcal{G}})$$

To prove the goal above, we start with an arbitrary interleaved history $h_{\mathcal{G}}$ s.t. $\mathcal{X}(h_{\mathcal{G}}) \subseteq S$ and convert it to a sequential history $h'_{\mathcal{G}}$ with the above properties. We perform this proof, by first creating the CCR partition of $h_{\mathcal{G}}$, $P = CCRPart(h_{\mathcal{G}})$, and then induct on the number of partitions in P that *are* interleaved.

Base Case: One interleaved CCR in P. Let $h^{ccr}_{\mathcal{G}} = P[i]$ be the interleaved history in $h_{\mathcal{G}}$. Now, let $h'_{\mathcal{G}} = h_{\mathcal{G}}[Seq_{|CCR}(h^{ccr}_{\mathcal{G}})/h^{ccr}_{\mathcal{G}}]$. Because $h^{ccr}_{\mathcal{G}}$ is the only interleaved sub-history in P and because of lemma E.3, we have that $h'_{\mathcal{G}}$ is a sequential history s.t. $\forall t, v_{\mathcal{G}}.\ \pi(h_{\mathcal{G}}, t) = \pi(h'_{\mathcal{G}}, t)$. Furthermore, because of lemma E.4 we have $\forall \sigma, v. M_{\mathcal{G}} \vdash (h_{\mathcal{G}}, v_{\mathcal{G}}, \sigma) \Downarrow \sigma' \Rightarrow M_{\mathcal{G}} \vdash (h'_{\mathcal{G}}, Seq(v_{\mathcal{G}}), \sigma) \Downarrow \sigma'$. Finally, because we have $M_{\mathcal{G}} \vdash (h_{\mathcal{G}}, v_{\mathcal{G}}, \sigma) \Downarrow \sigma'$ for some σ, this implies that $\mathsf{Expand}_{M_{\mathcal{G}}}(h, v_{\mathcal{G}}, \sigma) = (h'_{\mathcal{G}}, Seq(v_{\mathcal{G}}))$, which in turns implies $(h'_{\mathcal{G}}, v_{\mathcal{G}}) \backsim (h, Seq(v_{\mathcal{G}}))$.

Inductive Step. Next, we assume that our theorem holds for up to n interleaved CCRs in P, and will prove it for $n+1$. Similarly as above, we find the smallest i s.t. $P[i] = h^{ccr}_{\mathcal{G}}$ is an interleaved history. Again, we construct $h'_{\mathcal{G}} = h_{\mathcal{G}}[Seq_{|CCR}(h^{ccr}_{\mathcal{G}})/h^{ccr}_{\mathcal{G}}]$. Because of lemma E.3, we have that the number of interleaved histories in $CCRPart(h'_{\mathcal{G}})$ has strictly fewer number of interleaved sub-histories than P. Therefore, by our inductive hypothesis, we have that $(h'_{\mathcal{G}}, v'_{\mathcal{G}} \backsim (h, Seq(v_{\mathcal{G}}))$ for some history of h of M. This, combined with lemma E.4, proves that $(h_{\mathcal{G}}, v_{\mathcal{G}}) \backsim (h, Seq(v\mathcal{G}))$. □

E.3 Proof of Theorem 4.16

We now prove theorem 4.16 which states the correctness of our MaxSAT encoding.

Theorem 4.16. Let m be a model of the generated MaxSAT instance and $(\mathcal{L}, \mathcal{A}, \mathcal{P})$ be the synchronization protocol constructed as follows:

$$\mathcal{L} = \left\{v \mapsto \left\{l \mid m[h^l_v]\right\}\right\} \quad \mathcal{A} = \left\{\texttt{fld} \mid m[a_{fld}]\right\} \quad \mathcal{P} = \left\{p \mapsto l_i \mid IsWait(v,p), i = min(\{j \mid m[h^{l_j}_v]\})\right\}$$

where, $IsWait(v,p)$ is true if v is a `waituntil` statement on p. Then, $(\mathcal{L}, \mathcal{A}, \mathcal{P})$ is a correct synchronization protocol.

Proof. As mentioned earlier, a synchronization protocol must meet the following correctness criteria:

(1) If two fragments v_1, v_2 have a race (i.e., $\mathcal{R}(v_1, v_2) \neq \emptyset$), then the protocol must prevent this race with a lock or an atomic field.
(2) If a fragment interleaving $\chi = (v, e)$ is not safe, then the synchronization protocol must not allow fragment v to execute in between edge e.
(3) The protocol must be deadlock-free.

We show that, by construction, a model m returned by a MaxSAT solver always satisfies the above conditions.

(1) Model m prevents any races between two fragments because it must satisfy *all* hard constraints generated by rules Race-1 and Race-2 from Figure 8. Therefore, m will force two racy fragments to either share a lock or, when possible, convert all operations involving the racy field to equivalent atomic ones.
(2) Similarly, because model m must satisfy the hard constraints generated by rule I-Leave, any interleaving that was deemed unsafe by our static analysis is guaranteed to be infeasible in the resulting synchronization monitor.
(3) Finally, because of rules Wait and L-Order, the resulting synchronization monitor is guaranteed to be deadlock-free. Specifically, the hard constraints generated by rule L-Order enforce the invariant that all lock acquisitions respect the global lock order. Whereas, the hard constraints of rule Wait, enforce the same invariant for the translation of a `waituntil` statement into an equivalent statement in the target language (see Figure 16).

□

E.4 Proof of Theorem D.1

Finally, we prove the correctness of our monitor instrumentation procedure (Fig. 16).

Theorem E.5. *Let $\mathcal{S} = (\mathcal{L}, \mathcal{A}, \mathcal{P})$ be a synchronization protocol inferred over FGD $\mathcal{G} = (V, E)$ of input monitor M and M' be the result of procedure Instrument for M. Then, the following three conditions hold:*

(1) For every fragment $v \in V$, $l_i \in \mathcal{L}[v]$ iff fragment v holds lock l_i in M'

(2) If $i < j$, then l_i is never acquired whenever l_j is held.
(3) Field $f \in \mathcal{A}$ iff all its occurrences in M have been replaced with an atomic operation in M'.

Proof. All three conditions can be proved by providing certain guarantees for a subset of the rules of Figure 16. Note that operator $\rightsquigarrow$ (Figure 16) is guaranteed to visit every code fragment $v \in V$ in the FDG, since it recursively visits every element of the input monitor until it discovers *all* fragments of the given FDG. Next, we prove all three conditions.

Condition (1): For this condition, we need to prove that both fragments will only hold the locks required by the synthesized protocol $\mathcal{S}$. The logic of this proof depends on the number and type of predecessors of fragment v. We now present a case analysis:

Zero predecessors. This is the case of an entry fragment of a method in $\mathcal{G}$. Due to the structure of our input language and the definition of an FDG, this fragment *must* be a fragment that contains a single `waituntil` statement. The instrumentation of such a fragments is handled by rules Wait and Entry-Frag. Note, that rule Wait first calls Entry-Frag which acquires all locks needed by the fragmented defined by the `waituntil` fragment.

At least one predecessor. These types of fragments are handled by rules Wait, Branch-Frag, Reg-Frag-1, and Reg-Frag-2. All these rules maintain the following invariant for the fragment v that triggers them: before transferring control to any of v's successor, they release all locks needed by v but not needed by the successor (locksets of the form R_i) and acquire all locks needed by the successor but not held by v (locksets of the form A_i). This invariant combined with the fact that these are the only ways to transfer control flow in our input language, ensure that before executing a fragment in the output monitor all necessary locks (and only those) will be acquired.

Condition (2): This directly follows from:

(1) That procedure Instrument uses auxiliary relation Acq to instrument lock acquisitions, which as shown in Figure 16 does so in increasing order of lock indices.
(2) The guarantee provided by Theorem 4.16 that the synthesized protocol acquires locks in increasing order along every control-flow edge.

Condition (3): This condition is ensured by rule Frag-Stmt of Figure 16 that ensures oracle $\rightarrow_{\mathcal{A}}$ is called on every fragment of $\mathcal{G}$. □