Tractable Refinement Checking for Concurrent Objects

Ahmed Bouajjani
LIAFA, Université Paris Diderot
abou@liafa.univ-paris-diderot.fr

Michael Emmi
IMDEA Software Institute
michael.emmi@imdea.org

Constantin Enea Jad Hamza
LIAFA, Université Paris Diderot
{cenea,jhamza}@liafa.univ-paris-diderot.fr

Abstract

Efficient implementations of concurrent objects such as semaphores, locks, and atomic collections are essential to modern computing. Yet programming such objects is error prone: in minimizing the synchronization overhead between concurrent object invocations, one risks the conformance to reference implementations — or in formal terms, one risks violating *observational refinement*. Testing this refinement even within a single execution is intractable, limiting existing approaches to executions with very few object invocations.

We develop a polynomial-time (per execution) approximation to refinement checking. The approximation is parameterized by an accuracy $k \in \mathbb{N}$ representing the degree to which refinement violations are visible. In principle, more violations are detectable as k increases, and in the limit, all are detectable. Our insight for this approximation arises from foundational properties on the partial orders characterizing the happens-before relations between object invocations: they are *interval orders*, with a well defined measure of complexity, i.e., their *length*. Approximating the happens-before relation with a possibly-weaker interval order of bounded length can be efficiently implemented by maintaining a bounded number of integer counters. In practice, we find that refinement violations can be detected with very small values of k, and that our approach scales far beyond existing refinement-checking approaches.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification

General Terms Reliability, Verification

Keywords Concurrency; Refinement; Linearizability

1. Introduction

Efficient implementations of concurrent objects such as semaphores, locks, and atomic collections including stacks and queues are vital to modern computer systems. Programming them is however error prone. To minimize synchronization overhead between concurrent object-method invocations, implementors avoid blocking operations like lock acquisition, allowing methods to execute concurrently. However, concurrency risks unintended inter-operation interference, and risks conformance to reference implementations. Conformance is formally captured by *observational refinement*: given two libraries L_1 and L_2 implementing the methods of some concurrent object, we say L_1 *refines* L_2 if and only if every computation of every program using L_1 would also be possible were L_2 used instead.

Verifying observational refinement is intrinsically hard, and generally undecidable [4]. Here we develop a tractable algorithmic approach to detecting refinement violations, in two parts.

The first part, outlined in Section 1.1, establishes a foundational characterization of refinement in terms of sets of so-called *histories*. Histories abstract executions into the happens-before (partial) order between object-method invocations. We show that refinement between libraries is equivalent to a history-set inclusion problem. Consequently, violations correspond to excluded histories.

Since refinement-violation detection for even a single execution is NP-hard [8], the second part, outlined in Section 1.2, develops a novel approximation for refinement checking. We demonstrate that our approach is feasible, leading to scalable refinement-violation-detection algorithms. The insight behind our approximation exploits fundamental properties of the histories arising from shared-memory concurrent executions: they are a special class of partial orders called interval orders. Such orders admit convenient representations leading to efficient automation, and reveal a useful parameter for refining our approximation. In practice we find that even coarse approximations uncover refinement violations, and can be implemented much more efficiently than existing approaches.

1.1 Characterizing Observational Refinement

Naturally, automating observational refinement verification is challenging. The most immediate obstacle arises from the quantification over the infinitely-many possible library-client programs: a library L_1 refines another library L_2 if every observation of every client program using L_1 is also admitted using L_2. Our first contribution is to provide a *precise* characterization of refinement as a set-inclusion problem, defined independently from libraries' execution contexts, between the partial orders of operations admitted by each library. More precisely, we associate to each execution e a partial order $H(e)$ on its object-method invocations, called a *history*. An operation o_1 is considered to happen before an operation o_2 in $H(e)$ if o_1 completes before o_2 is invoked in e. We prove that a library L_1 refines another L_2 if and only if the set $H(L_1)$ of L_1's histories (i.e., associated with L_1's executions) is included in $H(L_2)$.

This characterization is a fundamental result that offers a fresh view for reasoning about refinement. Thus far, the principal approach for checking observational refinement in the literature is based on *linearizability* [11], requiring that operations of every execution of L_1 can be permuted into a serial execution of L_2 while preserving the return-call order between operations. While linearizability implies observational refinement [6], we demonstrate that the converse does not generally hold. To shed light on the subtle relationship between these concepts, we investigate the link between history inclusion and linearizability. We prove that when L_2 is atomic, which is often the case for reference implementations, history inclusion between L_1 and L_2, and therefore observational refinement, is equivalent to linearizability.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2677002

1.2 Approximating Observational Refinement

Besides offering a fresh perspective on observational refinement, our characterization leads to an efficient approximation-based approach for detecting refinement violations, exploiting fundamental properties of library executions and their histories. We consider a *weakening preorder* $\preceq$ on histories as a means for approximation: a history h_1 is *weaker* than another history h_2, written $h_1 \preceq h_2$, essentially if h_1 is obtained from h_2 by relaxing order constraints. We show that if a library L admits a history h, then L also admits every weaker history $h' \preceq h$. Our approximation principle considers weakenings $A(h) \preceq h$ of histories $h \in H(L_1)$, such that checking $A(h) \in H(L_2)$ is *tractable*. If $A(h) \notin H(L_2)$ then $h \notin H(L_2)$, and thus a violation is found.

The challenge is to design a parameterized approximation A_k, for $k \in \mathbb{N}$, such that $A_k(h) \in H(L_2)$ is decidable in *polynomial time* in the size of h. The approximation should also be *complete*, in the sense that for any history h, there exists $k \in \mathbb{N}$ such that $h \preceq A_k(h)$, ensuring that any violation can be captured with a large enough parameter value. Finally, the approximation should be easy to implement, and catch violations with small parameter values.

Our approximation scheme exploits a fundamental property of shared-memory library executions: their histories are *interval orders*, a special case of partial orders which admit *canonical representations* in which each operation o is mapped to a positive-integer-bounded interval $I(o)$. An operation o_1 happens before another operation o_2 if and only if the interval $I(o_1)$ ends before $I(o_2)$ begins [9]. Interval orders are equipped with a natural notion of *length*, which corresponds to the smallest integer constant for which an interval mapping exists. Our approximation A_k maps each history h to a weaker history $A_k(h)$ of interval length at most k.

Bounded-interval-length orders admit a convenient representation of histories using counting: each interval is represented by a counter whose value reflects the number of operations spanning that interval. Such a representation conveniently opens the door to symbolic manipulation of history sets using arithmetic constraints, which we describe using a simple *operation counting logic (OCL)*. We demonstrate that this logic has a polynomial-time model-checking problem, and is suitable for reasoning about commonly-used concurrent object libraries including atomic collections like stacks and queues. We also demonstrate that OCL formulæ can be systematically constructed for a broad class of concurrent objects.

Moreover, counting-based representations can be efficiently monitored in polynomial-time and space using simple counter increments and decrements at the precise moments when operations begin and end. By maintaining a k-interval-length approximation $A_k(H(e))$ of an execution e, we effectively reduce our approximate refinement-checking problem to a *safety verification* problem by periodically checking whether $A_k(H(e)) \models \Psi$, where Ψ is an OCL formula characterizing the k-interval-length histories of $H(L_2)$.

Empirically, we demonstrate that our approach is effective in both dynamic-checking and static-checking contexts. Used in the dynamic setting, we maintain relatively-low runtime overhead which does not increase in the length of executions. Our approach scales far beyond the existing approaches based on linearizability, in which enumerating linearizations explodes exponentially in the length of executions. Furthermore, our approximation A_k is well-suited for catching refinement violations with small parameter values: violations are most often detected with $k \leq 2$, and almost always with $k \leq 4$. In fact, we even prove that many violations to atomic collection objects including stacks and queues can always be caught with $k \leq 3$. In the static-checking context, our counting-based representation allows us to leverage off-the-shelf SAT/SMT-based symbolic program exploration tools including CSeq [7] (with backend CBMC [13]) and SMACK [24] (with backend Corral [15]) to discover refinement violations.

```
struct node *Top;

void push(int v):
  struct node *n,*t;
  n = malloc(sizeof( *n));
  n->data = v;
  do {
    struct node *t = Top;
    n->next = t;
  } while (! CAS (&Top, t, n));

int pop():
  struct node *n,*t;
  do {
    *t = Top;
    if (t==NULL) return EMPTY;
    n = t->next;
  } while (! CAS (&Top, t, n))
  int result = t->data;
  free(t);
  return result;

struct node {
  int data;
  struct node *next;
}

void Thread1():
  push(1);
  int x = pop();

void Thread2():
  int y = pop();
  push(2);
  push(3);
  int z = pop();
```

Figure 1. An implementation of Treiber's stack. The `pop` operation returns the value `EMPTY` when the stack is empty.

1.3 Summary of Contributions

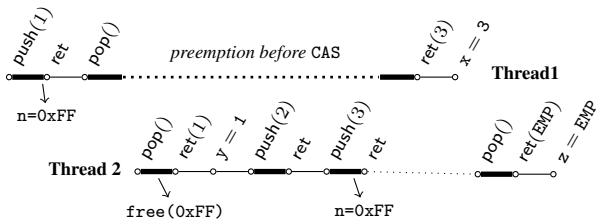
This work makes the following contributions:

- A characterization of observational refinement as a history-set-inclusion problem (§4), and a proof of equivalence with linearizability for atomic reference implementations (§5).
- An under-approximation for detecting observational refinement violations based on a weakening preorder on histories, exploiting the fact that library histories are interval orders (§6).
- An efficient implementation of our approximation using counters. Specifically, we reduce refinement checking to safety-property checking (§6.2) using symbolic arithmetic representations of bounded-interval-length history sets (§6.1), demonstrate cut-off bounds for atomic-collection objects (§7), and develop an automatic construction of counting representations (§8).
- Experimental validation of our approximation-based approach in both dynamic-checking and static-checking settings (§9).

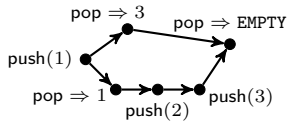
2. Motivating Example

Figure 1 lists a non-blocking Treiber's stack [26] implementation providing `push` and `pop` methods. This implementation stores pushed elements into a singly-linked list rooted at `Top`, and avoids blocking lock acquisitions in favor of non-blocking compare-and-swap (`CAS`) operations in order to maximize parallelism, allowing methods to interleave their internal actions. In one atomic step, the `CAS` operation assigns `Top = n` only if `Top == t`.

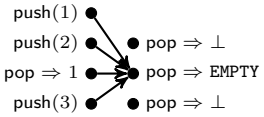
Unfortunately this implementation suffers from a subtle concurrency bug, now commonly known as an "ABA" bug [17]. The bug manifests in the two-thread program of Figure 1, via the execution depicted in Figure 2(a). Essentially, Thread 1 wrongfully assumes the absence of interference from other threads on the successful `CAS` operation. Thread 1 is preempted right before executing its `CAS` in the `pop` method; at that moment, its `t` variable points to the first element in the list at address `0xFF` added by `push(1)`, and `n == NULL`. While Thread 2 updates the list with two additional elements, added by `push(2)` and `push(3)`, the `t` variable of Thread 1 still points to the list's first element at address `0xFF`, which was freed by Thread 2's call to `pop`, and reallocated in the call to `push(3)`. When Thread 1 resumes, its `CAS` succeeds, effectively removing two elements from the list instead of one. The final `pop` of Thread 2 thus erroneously returns `EMPTY`. Intuitively, this is a problem because the `EMPTY` value should not have been returned since more elements have been pushed than popped prior to Thread 2's final `pop` operation.



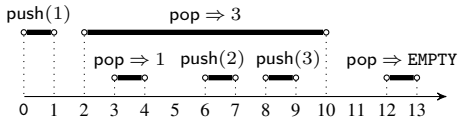
(a) An execution e of the program; it depicts calls, returns, and assignments, and time progresses from left to right.



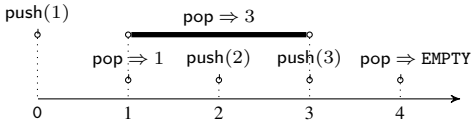
(b) The history $H(e)$ of execution e.



(c) A history weaker than $H(e)$.



(d) The history $H(e)$ as an interval order.



(e) The canonical representation of $H(e)$.

Figure 2. An execution and its history.

This bug exposes the fact that our CAS-based implementation does not conform to programmers' expectations of a stack object whose operations execute atomically, e.g., by holding a lock for the duration of each operation. In particular, the assignment `z = EMPTY` should never have occurred.

This idea of conformance is rigorously captured by the formal notion of *observational refinement*. Essentially, an implementation L_1 of a concurrent object *refines* another implementation L_2 if every observable behavior of a program using L_1 is also observable using L_2. This property clearly does not hold between the CAS-based implementation of Figure 1 and a correct atomic lock-based implementation, since `y = 1; x = 3; z = EMPTY` is observable using the CAS-based implementation, yet not using the atomic one.

3. Observational Refinement

We formalize the criterion of *observational refinement* using a simple yet universal model of computation, namely labeled transition systems (LTS). This model captures shared-memory programs with an arbitrary number of threads, abstracting away the details of any particular programming system irrelevant to our development.

A *labeled transition system* $A = (Q, \Sigma, q_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $q_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $e \in \Sigma^*$ is denoted e_i. An *execution* of A is a sequence $e \in \Sigma^*$ such that for some $q_1, q_2, \ldots, q_{|e|} \in Q$, we have $\delta(q_i, e_i, q_{i+1})$ for each i such that $0 \leq i < |e|$. The projection $e|\Gamma$ is the maximum subsequence of e over alphabet Γ. $E(A)$ denotes the set of A's executions, and $E(A)|\Gamma$ their projections over Γ (note that $E(A)$ is prefix closed). The *synchronous product* $A_1 \times A_2$ of two LTSs is defined as usual, respecting $E(A_1 \times A_2)|(\Sigma_1 \cap \Sigma_2) = E(A_1)|\Sigma_2 \cap E(A_2)|\Sigma_1$.

3.1 Libraries

Programs interact with libraries by calling named library *methods*, which receive *parameter values* and yield *return values* upon completion. We fix arbitrary sets $\mathbb{M}$ and $\mathbb{V}$ of method names and parameter/return values.

Example 3.1. *The method and value sets for the stack implementation in Figure 1 are* $\mathbb{M} = \{\mathsf{push}, \mathsf{pop}\}$ *and* $\mathbb{V} = \mathbb{N} \cup \{\mathtt{EMPTY}\}$.

We fix an arbitrary set $\mathbb{O}$ of operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$$C = \{m(v)_o : m \in \mathbb{M}, v \in \mathbb{V}, o \in \mathbb{O}\}, \text{ and}$$
$$R = \{\mathsf{ret}(v)_o : v \in \mathbb{V}, o \in \mathbb{O}\}$$

of *call actions* and *return actions*; each call action $m(v)_o$ combines a method $m \in \mathbb{M}$ and value $v \in \mathbb{V}$ with an *operation identifier* $o \in \mathbb{O}$. Operation identifiers are used to pair call and return actions. We denote the operation identifier of a call/return action a by $\mathsf{op}(a)$. Call and return actions $c \in C$ and $r \in R$ are *matching*, written $c \mapsto r$, when $\mathsf{op}(c) = \mathsf{op}(r)$. A word $e \in \Sigma^*$ over alphabet Σ, such that $(C \cup R) \subseteq \Sigma$, is *well formed* when:

- Each return is preceded by a matching call:
  $e_j \in R$ implies $e_i \mapsto e_j$ for some $i < j$.
- Each operation identifier is used in at most one call/return:
  $\mathsf{op}(e_i) = \mathsf{op}(e_j)$ and $i < j$ implies $e_i \mapsto e_j$.

We say that the well-formed word $e \in \Sigma^*$ is *sequential* when

- Operations do not overlap:
  $e_i, e_k \in C$ and $i < k$ implies $e_i \mapsto e_j$ for some $i < j < k$.

Well-formed words represent executions. We assume every set of well-formed words is closed under isomorphic renaming of operation identifiers. For notational convenience, we often associate $\mathbb{O}$ with $\mathbb{N}$, e.g., writing $m(u)_1$ and $\mathsf{ret}(v)_2$ in place of $m(u)_{o_1}$ and $\mathsf{ret}(v)_{o_2}$. An operation o of an execution e is *completed* when both call and return actions $m(u)_o$ and $\mathsf{ret}(v)_o$ of o occur in e, and is otherwise *pending*.

Example 3.2. *The well-formed words*

$\mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{pop}_3\ \mathsf{ret}_1\ \mathsf{ret}(0)_3\ \mathsf{ret}(0)_2$, *and* $\mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{pop}_3\ \mathsf{ret}_1\ \mathsf{ret}(0)_2$

represent executions in which one call to the $\mathsf{push}(0)$ *method overlaps with two calls to* pop. *In the first execution both calls to* pop *have matching return actions* $\mathsf{ret}(0)$*, i.e., the operations* 2 *and* 3 *are completed, while operation* 3 *is pending in the second, it has no matching return.*

Libraries dictate the execution of methods between their call and return points. Accordingly, a library cannot prevent a method from being called, though it can decide not to return. Furthermore, any library action performed in the interval between call and return points can also be performed should the call have been made earlier, and/or the return made later. Our technical results rely on these properties. A library thus allows any sequence of invocations to its methods made by *some* program.

Definition 3.1. *A* library L *is an LTS over alphabet* $C \cup R$ *such that each execution* $e \in E(L)$ *is well formed, and*

- *Call actions* $c \in C$ *cannot be disabled:*
  $e \cdot e' \in E(L)$ *implies* $e \cdot c \cdot e' \in E(L)$ *if* $e \cdot c \cdot e'$ *is well formed.*
- *Call actions* $c \in C$ *cannot disable other actions:*
  $e \cdot a \cdot c \cdot e' \in E(L)$ *implies* $e \cdot c \cdot a \cdot e' \in E(L)$.
- *Return actions* $r \in R$ *cannot enable other actions:*
  $e \cdot r \cdot a \cdot e' \in E(L)$ *implies* $e \cdot a \cdot r \cdot e' \in E(L)$.

We write $e_1 \rightsquigarrow e_2$ when e_2 can be derived from e_1 by applying zero or more of the above rules. The *closure* of a set E of executions under $\rightsquigarrow$ is denoted $\overline{E}$.

Note that even a library that implements *atomic methods*, e.g., by guarding method bodies with a global-lock acquisition, admits executions in which method calls and returns overlap. A library which accesses the client's thread identifiers can be modeled by taking thread identifiers as method parameters.

Example 3.3. *Any library which admits the execution*

$$\mathsf{push}(0)_1\ \mathsf{ret}_1\ \mathsf{pop}_2\ \mathsf{ret}(0)_2$$

with sequential calls to push *and* pop *must also admit*

$$\mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{ret}_1\ \mathsf{ret}(0)_2 \text{ and } \mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{pop}_3\ \mathsf{ret}_1\ \mathsf{ret}(0)_2,$$

among others, yet need not admit an execution

$$\mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{pop}_3\ \mathsf{ret}_1\ \mathsf{ret}(0)_3\ \mathsf{ret}(0)_2$$

with two completed pop *operations returning* 0.

A library L is called *atomic* if it is defined by the closure of some set E of sequential executions, i.e., $E(L) = \overline{E}$. When such a set E exists, it is unique, and we call it *the kernel of* L, denoted by $\ker L$. Note that $\ker L$ contains only completed operations since $e_1 \cdot e_2 \rightsquigarrow e_1 \cdot c \cdot e_2$, for any unmatched call c. Atomic libraries are often considered as specifications for concurrent objects.

Example 3.4. *The* atomic stack *is the library whose kernel is the set of sequential executions for which the return value of each* pop *operation is either*

- *the argument value v to the last unmatched* push *operation, or*
- `EMPTY` *if there are no unmatched* push *operations.*

In practice, the atomic stack can be implemented by guarding the methods of a "sequential" stack object by global-lock acquisition.

3.2 Refinement between Libraries

Refinement between libraries is defined with respect to the observable actions of programs which invoke library methods. Complementary to libraries, programs control their execution outside of method call and return points. Accordingly, any program action performed in the interval between call and return points can also be performed should the call have been made later, and/or the return made earlier. A program thus allows any sequence of matching returns generated by *some* implementation of the methods it invokes.

Definition 3.2. *A* program P *over actions* Σ *is an LTS over alphabet* $(\Sigma \uplus C \uplus R)$ *where each execution* $e \in E(P)$ *is well formed, and*

- *Call actions $c \in C$ cannot enable other actions:* $e \cdot c \cdot a \cdot e' \in E(P)$ *implies* $c \mapsto a$ *or* $e \cdot a \cdot c \cdot e' \in E(P)$.
- *Return actions $r \in R$ cannot disable other actions:* $e \cdot a \cdot r \cdot e' \in E(P)$ *implies* $a \mapsto r$ *or* $e \cdot r \cdot a \cdot e \in E(P)$.
- *Return actions $r \in R$ cannot be disabled:* $e \cdot e' \in E(P)$ *implies* $e \cdot r \cdot e' \in E(L)$ *if* $e \cdot r \cdot e'$ *is well formed.*

Example 3.5. *Any program which admits the execution*

$$\mathsf{push}(0)_1\ \mathsf{pop}_2\ \mathsf{ret}(0)_2\ \mathsf{pop}_3\ \mathsf{ret}_1,$$

with two sequential pop *calls concurrent with* push*, must also admit*

$$\mathsf{push}(0)_1\ \mathsf{ret}_1\ \mathsf{pop}_2\ \mathsf{ret}(0)_2\ \mathsf{pop}_3 \quad \textit{and}$$
$$\mathsf{push}(0)_1\ \mathsf{ret}_1\ \mathsf{pop}_2\ \mathsf{ret}(0)_2\ \mathsf{pop}_3\ \mathsf{ret}(\mathtt{EMPTY})_3,$$

among others, in which all three calls are sequential and the second pop *may return (with any value), yet need not admit an execution*

$$\mathsf{push}(0)_1\ \mathsf{ret}_1\ \mathsf{pop}_3\ \mathsf{pop}_2\ \mathsf{ret}(0)_2,$$

in which the two calls to pop *are concurrent. The set of executions admitted by a program allows any possible implementation of the methods. While programs cannot force methods to execute concurrently, they can force methods to execute sequentially, e.g., by waiting for one to return before calling the next.*

Refinement between libraries L_1 and L_2 means that any program execution possible with L_1 is also possible with L_2.

Definition 3.3. *The library* L_1 refines L_2*, written* $L_1 \leq L_2$*, iff*

$$E(P \times L_1)|\Sigma \ \subseteq\ E(P \times L_2)|\Sigma$$

for all programs P *over actions* Σ.

Note that $\leq$ is a preorder over libraries. As library and program alphabets only intersect on call and return actions $C \cup R$, our formalization supposes that programs and libraries communicate only through method calls and returns, and not, e.g., through shared random-access memory.

Example 3.6. *The incorrect Treiber's stack implementation of Figure 1 does not refine an atomic lock-based reference implementation, since the execution of Figure 2 is admitted by its composition with the two-thread program of Figure 1.*

4. History Inclusion

Though we seek to develop automated techniques to check observational refinement between libraries, the definition of Section 3 does not suggest any practical means; it only suggests enumerating every possible execution of every possible program. In this section we introduce an equivalent notion based on concise abstractions of program executions called *histories*. Besides being independent of programs, this equivalent notion helps expose the structure of the refinement problem, and suggests practical means of automation which we develop in Section 6.

4.1 Histories

For given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix a set $\mathbb{L} = \mathbb{M} \times \mathbb{V} \times (\mathbb{V} \cup \{\bot\})$ of *operation labels*, and denote the label $\langle m, u, v\rangle$ by $m(u) \Rightarrow v$. A *history* $h = \langle O, <, f\rangle$ is a partial order $<$ on a set $O \subseteq \mathbb{O}$ of operation identifiers labeled by $f : O \to \mathbb{L}$ for which $f(o) = m(u) \Rightarrow \bot$ implies o is maximal in $<$. The *history* $H(e)$ of a well-formed execution $e \in \Sigma^*$ labels each operation with a method-call summary, and orders non-overlapping operations:

- $O = \{\mathsf{op}(e_i) : 0 \leq i < |e| \text{ and } e_i \in C\}$,
- $\mathsf{op}(e_i) < \mathsf{op}(e_j)$ iff $i < j$, $e_i \in R$, and $e_j \in C$.
- $f(o) = \begin{cases} m(u) \Rightarrow v & \text{if } m(u)_o \in e \text{ and } \mathsf{ret}(v)_o \in e \\ m(u) \Rightarrow \bot & \text{if } m(u)_o \in e \text{ and } \mathsf{ret}(\_)_o \notin e \end{cases}$

An operation of h labeled by $\ell \in \mathbb{L}$ is called an ℓ *operation*. The histories admitted by L are $H(L) = \{H(e) : e \in E(L)\}$.

Example 4.1. *Figure 2(b) depicts the history of the execution in Figure 2(a). Arrows depict the order relation modulo transitivity. Operations o_1 and o_2 are ordered in h if o_1's return precedes o_2's call. For example,* $\mathsf{push}(1)$ *precedes* $\mathsf{pop} \Rightarrow 3$*. However,* $\mathsf{pop} \Rightarrow 1$ *is incomparable to* $\mathsf{pop} \Rightarrow 3$ *because* $\mathsf{pop} \Rightarrow 1$*'s return comes after* $\mathsf{pop} \Rightarrow 3$*'s call, and* $\mathsf{pop} \Rightarrow 3$*'s return comes after* $\mathsf{pop} \Rightarrow 1$*'s call. The order among operations' call actions is irrelevant, as is the order among their return actions.*

While the general concept of histories allows arbitrary partial orders of operations, any history $H(e)$ arising from an LTS execution e falls into a restricted class called *interval orders*. Intuitively, this is because our execution model assumes that operations share a common notion of global time: the actions in an execution are linearly ordered.

Definition 4.1. *An* interval order *is a partial order* $\langle O, <\rangle$ *such that* $o_1 < o_3$ *and* $o_2 < o_4$ *implies* $o_1 < o_4$ *or* $o_2 < o_3$.

Lemma 4.1. *The history $H(e) = \langle O, f, < \rangle$ of a well-formed execution e forms an interval order $\langle O, < \rangle$.*

Proof. Suppose $o_1 < o_3$ and $o_2 < o_4$ in $H(e)$, and fix i_1, i_2, i_3, i_4 such that e_{i_1} and e_{i_2} are the return actions of o_1 and o_2, and e_{i_3} and e_{i_4} are the call actions of o_3 and o_4; note that $i_1 < i_3$ and $i_2 < i_4$. Since $<$ linearly orders $\{i_1, i_2, i_3, i_4\}$, either $i_1 < i_4$, in which case $o_1 < o_4$, or $i_4 < i_1$, in which case $i_2 < i_4 < i_1 < i_3$, so $o_2 < o_3$. □

Example 4.2. *Figure 2(d) pictures the history in Figure 2(b) as an interval order. Each operation is represented by an integer-bounded interval on the real number line such that $o_1 < o_2$ iff the interval associated to o_1 finishes before the interval associated to o_2.*

In this work we consider only histories of well-formed executions, i.e., those forming interval orders, without explicit qualification.

This notion of histories gives rise to a natural order relating histories by their operation ordering. Basically, a history h_1 is *weaker* than another history h_2 if h_2 contains all completed operations of h_1, and preserves the order between h_1's operations. The pending operations of h_1 can be either omitted or completed in h_2.

Definition 4.2. *Let $h_1 = \langle O_1, <_1, f_1 \rangle$ and $h_2 = \langle O_2, <_2, f_2 \rangle$. We say h_1 is* weaker than *h_2, written $h_1 \preceq h_2$, when there exists an injection $g : O_2 \to O_1$ such that*

- *$o \in \mathrm{range}(g)$ when $f_1(o) = m(u) \Rightarrow v$ and $v \neq \bot$,*
- *$g(o_1) <_1 g(o_2)$ implies $o_1 <_2 o_2$ for each $o_1, o_2 \in O_2$,*
- *$f_1(g(o)) \ll f_2(o)$ for each $o \in O_2$.*

where $(m_1(u_1) \Rightarrow v_1) \ll (m_2(u_2) \Rightarrow v_2)$ iff $m_1 = m_2$, $u_1 = u_2$, and $v_1 \in \{v_2, \bot\}$. We say h_1 and h_2 are equivalent *when $h_1 \preceq h_2$ and $h_2 \preceq h_1$.*

Example 4.3. *Figure 2(c) pictures a history h' weaker than the history h in Figure 2(b). Note that h' contains two pending $\mathsf{pop} \Rightarrow \bot$ operations, one of them is completed in h (it corresponds to the $\mathsf{pop} \Rightarrow 3$ operation) and one of them is omitted in h.*

Throughout this work we do not distinguish between equivalent histories, and we assume every set H of histories is closed under inclusion of equivalent histories, i.e., if h_1 and h_2 are equivalent and $h_1 \in H$, then $h_2 \in H$ as well.

4.2 History Inclusion is Equivalent to Refinement

Our notion of histories has two important properties which makes refinement between libraries equivalent to inclusion between their history sets. The first property is that libraries are closed under the weakening relation $\preceq$. Essentially when L admits the history $H(e)$ of an execution e, then it also admits the history $H(e')$ of any e' which is $\rightsquigarrow$-derivable from e. Our definitions of $\rightsquigarrow$ and $\preceq$ coincide to imply $H(e') \preceq H(e)$.

Lemma 4.2. *If $h_1 \in H(L)$ and $h_2 \preceq h_1$ then $h_2 \in H(L)$.*

A history h is essentially an abstraction of the set $\{e : H(e) = h\}$ of executions which preserve an order between returns preceding calls. The closure properties of Definition 3.1 ensure that a library L admits all executions e with $H(e) \in H(L)$.

Lemma 4.3. $E(L) = \{e \in (C \cup R)^* : H(e) \in H(L)\}$.

Lemmas 4.2 and 4.3 ultimately imply the equivalence between refinement and history inclusion. Essentially, any given history h of a library L_1 can be captured by a program P_h whose observations witness the ordering among h's operations. However, its observations cannot forbid additional orderings between h's operations (this is due to the closure properties in Definition 3.2). The refinement $L_1 \leq L_2$ implies that L_2 also admits those observations, through one or more possibly-stronger histories h'. It then follows from Lemma 4.2 that L_2 also admits $h \preceq h'$. For the reverse direction, it follows from Lemma 4.3 that history inclusion implies inclusion of executions, and ultimately observations.

Theorem 1. *$L_1 \leq L_2$ iff $H(L_1) \subseteq H(L_2)$.*

Proof. ($\Rightarrow$) Let $h = \langle O, <, f \rangle \in H(L_1)$; we show $h \in H(L_2)$ by constructing a program P_h over actions Σ which only admits executions with histories stronger than h:

$$\forall e \in E(P_h).\ |(e|\Sigma)| = n \implies h \preceq H(e),$$

where $n = |\{o \in O : f(o) = m(u) \Rightarrow v \neq \bot\}|$ is the number of completed operations in h. Given such a program P_h, taking any execution $e_1 \in E(P_h \times L_1)$ with $|(e_1|\Sigma)| = n$, we must also have an execution $e_2 \in E(P_h \times L_2)$ such that $(e_2|\Sigma) = (e_1|\Sigma)$ by definition of $L_1 \leq L_2$. Since $|(e_2|\Sigma)| = n$ and $e_2 \in E(P_h)$, we also know that $h \preceq H(e_2)$, and since $(e_2|C \cup R) \in E(L_2)$, we have $H(e_2) \in H(L_2)$, along with any history weaker than $H(e_2)$ by Lemma 4.2, namely h.

We construct $P_h = \langle Q, \Sigma, q_0, \delta \rangle$ over alphabet $\Sigma = C \cup R \cup \{a\}$ whose states $Q : O \to \mathbb{B}^2$ track operations called/completed status. The initial state is $q_0 = \{o \mapsto \langle \bot, \bot \rangle : o \in O\}$. Transitions are given by,

for each $q \in Q, o \in O, m \in \mathbb{M}, v \in \mathbb{V}$

if $f(o) = m(v) \Rightarrow \_$ and $q(o')$ for all $o' < o$ then

$q[o \mapsto \bot, \bot] \xrightarrow{m(v)_o} q[o \mapsto \top, \bot]$

if $f(o) = m(\_) \Rightarrow v$ then

$q[o \mapsto \top, \bot] \xrightarrow{\mathsf{ret}(v)_o} \cdot \xrightarrow{a} q[o \mapsto \top, \top]$

if $f(o) = m(\_) \Rightarrow \bot$ then

$q[o \mapsto \top, \bot] \xrightarrow{\mathsf{ret}(v)_o} q[o \mapsto \top, \top]$

It is routine to verify that P_h is a program according to Definition 3.2. Moreover, in any execution $e \in E(P_h)$, the call $m(u)_o$ of any operation o must come after the return $\mathsf{ret}(v)_{o'}$ of each $o' < o$. Furthermore, all completed operations of h are completed in e if and only if $(e|\Sigma) = a^n$. Note that e may contain more completed operations than h (because of the last transition rule) but less pending operations (because the transition rule corresponding to the call of a pending operation may not have been applied in e). It follows that $|(e|\Sigma)| = n \implies h \preceq H(e)$.

($\Leftarrow$) Let P be a program over actions Σ, and $e \in E(P \times L_1)$; we show that $e \in E(P \times L_2)$. Since $(e|C \cup R) \in E(L_1)$, we know $H(e) \in H(L_1)$ by definition of $H(L_1)$, and then $H(e) \in H(L_2)$ by hypothesis. By Lemma 4.3 we deduce $(e|C \cup R) \in E(L_2)$, and thus by definition of LTS composition, $e \in E(P \times L_2)$. □

5. Comparison with Linearizability

The linearizability criterion [11] provides an alternative characterization of library conformance which implies observational refinement [6]. In this section we demonstrate that linearizability is generally stricter than observational refinement, yet the two criteria coincide when the reference implementation is atomic.

Linearizability [11] is defined by an execution order: $e_1 \sqsubseteq e_2$ iff there exists a well-formed execution e_1' obtained from e_1 by appending return actions, and deleting call actions, such that:

> e_2 is a permutation of e_1' that preserves the order between return and call actions, i.e., a given return action occurs before a given call action in e_1' iff the same holds in e_2.

For example, the second and the third executions in Example 3.3 are $\sqsubseteq$-smaller than the first (with sequential calls to push and pop).

An execution e_1 is *linearizable* w.r.t. a library L_2[1] iff there exists a sequential execution $e_2 \in E(L_2)$, with only completed operations, such that $e_1 \sqsubseteq e_2$. A library L_1 is *linearizable* w.r.t. L_2, written $L_1 \sqsubseteq L_2$, iff each execution $e_1 \in E(L_1)$ is linearizable w.r.t. L_2.

Since linearizability compares executions of L_1, which may contain pending operations, with executions of L_2, in which every operation is completed, observational refinement need not imply linearizability when L_2 contains non-terminating methods, i.e., where the calls to these methods are pending in all executions.

Example 5.1. *Let L be the library whose kernel contains the single execution $e = m(u)_1\ m'(u)_2\ \mathsf{ret}(v)_1$, in which the call to m' is pending. Although L refines itself, since refinement is reflexive, L is not linearizable w.r.t. itself, since e could only be linearizable w.r.t. L if $E(L)$ were to contain one of the following executions:*

$$m(u)_1\ \mathsf{ret}(v)_1 \quad m(u)_1\ m'(u)_2\ \mathsf{ret}(v)_1\ \mathsf{ret}(\_)_2$$

$$m(u)_1\ \mathsf{ret}(v)_1\ m'(u)_2\ \mathsf{ret}(\_)_2 \quad m'(u)_2\ \mathsf{ret}(\_)_2 m(u)_1\ \mathsf{ret}(v)_1.$$

Yet $E(L) = \overline{\{e\}}$ clearly contains none of them.

A typical correctness criterion for concurrent objects is linearizability with respect to atomic versions of themselves. Despite the negative general result of Example 5.1, when restricted to atomic libraries, linearizability and history inclusion, and thus observational refinement, coincide. This relationship essentially follows from the relationship between the order relations $\sqsubseteq$ and $\preceq$.

Lemma 5.1. *$e_1 \sqsubseteq e_2$ iff $H(e_1) \preceq H(e_2)$.*

Proof. By definition $e_1 \sqsubseteq e_2$ means that there exists e_1' such that e_2 is a permutation of e_1' preserving the order between return and call actions, thus $e_2 \rightsquigarrow e_1'$, and thus $H(e_1') \preceq H(e_2)$. Furthermore, since e_1' is obtained from e_1 by appending returns and deleting calls, $H(e_1) \preceq H(e_1')$. By transitivity, $H(e_1) \preceq H(e_2)$. The other direction is equally straightforward. □

Theorem 2. *$L_1 \sqsubseteq L_2$ iff $H(L_1) \subseteq H(L_2)$, if L_2 is atomic.*

Proof. ($\Rightarrow$) Let $h \in H(L_1)$. By hypothesis, any execution e_1 with $H(e_1) = h$ is linearizable w.r.t. L_2, i.e., there exists an execution $e_2 \in L_2$ with only completed operations such that $e_1 \sqsubseteq e_2$. By Lemma 5.1, this implies $H(e_1) \preceq H(e_2)$. By the closure property in Lemma 4.2, if $H(e_2) \in H(L_2)$ then $h = H(e_1) \in H(L_2)$.

($\Leftarrow$) Let e_1 be an execution of L_1. By hypothesis, $H(e_1) \in H(L_2)$, which by Lemma 4.3, implies $e_1 \in E(L_2)$. Since L_2 is an atomic library, there exists a sequential execution $e_2 \in \ker E(L_2)$ with only completed operations such that $e_1 \sqsubseteq e_2$. Thus, e_1 is linearizable w.r.t. L_2. □

6. Approximating History Inclusion

By the equivalences of Section 5, checking whether a given history $H(e)$ is included in a set $H(L)$ of library histories is equivalent to checking whether $H(e)$ is linearizable with respect to L, for a atomic library L. It follows that deciding $H(e) \in H(L)$ is NP-hard for an arbitrary library L, since it is NP-hard for the atomic register object [8]. Generally speaking, the only known algorithms to decide $H(e) \in H(L)$ must check whether each possible linearization of the partially-ordered history $H(e)$ is equivalent to some sequential execution of operations according to L, backtracking to try alternate linearizations on each failed attempt. Recent work implies that the more general problem of checking whether *all histories* $H(L_1)$ of a given library L_1 are included in the set of histories $H(L_2)$ of a fixed library L_2 is undecidable, since it is equivalent to checking whether L_1 is linearizable w.r.t. L_2 when L_2 is atomic [4].

These complexity obstacles suggest investigating approximations to the history inclusion problems — i.e., both $h \in H(L)$ and its more general variation $H(L_1) \subseteq H(L_2)$ — in order to devise tractable algorithms.

In this work, we focus on parameterized *under approximations* for detecting violations to observational refinement, achieving increasing accuracy with decreasing efficiency. For this, we design a notion of parameterized history-weakening approximation functions A_k which map any history h to a weaker history $A_k(h) \preceq h$, and which have the following properties:

Strength-increasing: $A_0(h) \preceq A_1(h) \preceq \ldots \preceq A_k(h) \preceq h$.

Completeness: there exists $k \in \mathbb{N}$ such that $h \preceq A_k(h)$.

Tractable inclusion: $A_k(h) \in H(L)$ is decidable in polynomial time when k is fixed.

This weakening-based approximation is convenient since whenever $A_k(h)$ is not included in $H(L)$, then neither is h, since $H(L)$ is closed under weakening; if h were to belong to $H(L)$, then any weakening, and in particular $A_k(h)$, would also belong to $H(L)$. While completeness means that increasing k increases the ability to detect observational refinement violations, this must incur a decrease in efficiency since the inclusion problem $A_k(h) \in H(L)$ is NP-hard when k is not fixed. By design, the approximation function A_k allows us to solve the approximate history inclusion problem $A_k(h) \in H(L)$ in polynomial time for fixed k. For the more general problem of refinement between libraries L_1 and L_2, our approximation asks whether $A_k(h) \in H(L_1) \setminus H(L_2)$, and becomes decidable for fixed k, so long as the set $\{A_k(h) : h \in H(L_1)\}$ is computable. Completeness of A_k ensures overall completeness, i.e., that for any $h \in H(L_1) \setminus H(L_2)$ there is some $k \in \mathbb{N}$ such that $A_k(h) \in H(L_1) \setminus H(L_2)$.

Our key challenge is to develop approximation functions A_k for which history inclusion can be computed in polynomial time for fixed k, and for which observational refinement violations surface with small k. We demonstrate the latter in Section 7–9.

In this section we develop a schema of approximation functions for which the approximate history inclusion problem is polynomial-time computable. Our development exploits structural aspects of the history inclusion problem; in particular, we exploit the fact that histories are *interval orders*, with a natural measure of complexity, i.e., the interval order *length* [9]. Leveraging this notion of length, we abstract each history h to a weaker history $A_k(h)$ whose length is bounded by k, and represent the set $H(L)$ of histories, restricted to interval length k, by a formula against which $A_k(h)$ can be evaluated in polynomial time (§6.1). Finally, we exhibit a program monitoring scheme which can be used to decide our approximate observational refinement problem $\exists h.\ A_k(h) \in H(L_1) \setminus H(L_2)$, or as a general-purpose runtime-execution monitor (§6.2).

6.1 Bounded-Interval-Length History Inclusion

The *past* of an element $o \in O$ of a poset $\langle O, < \rangle$ is the set

$$\mathrm{past}(o) = \{o' \in O : o' < o\}$$

of elements ordered before o.

Example 6.1. *In the history h from Figure 2(b), the $\mathsf{pop} \Rightarrow 3$ and $\mathsf{pop} \Rightarrow 1$ operations have the same past, namely the operation $\mathsf{push}(1)$, while the past of the $\mathsf{push}(3)$ operation consists of the $\mathsf{push}(1)$, $\mathsf{pop} \Rightarrow 1$, and $\mathsf{push}(2)$ operations.*

This notion of operations' pasts induces a linear notion of time into execution histories due to the following fact.

[1] The original definition of Herlihy and Wing [11] assumes that L_2 is a set of sequential executions. Here we consider a slight extension adapted to concurrent executions, faithful to the original intention, where ker L_2 may be a set of sequential executions.

Lemma 6.1 (Rabinovitch [23]). *The set* $\{\mathrm{past}(o) : o \in O\}$ *of pasts of an interval order* $\langle O, <\rangle$ *is linearly ordered by set inclusion.*

Furthermore, this linear notion of time has an associated notion of *length*, which corresponds to the length of the linear order on operation's pasts.

Definition 6.1 (Greenough [9]). *The* length *of an interval order* $\langle O, <\rangle$ *is one less than the number of its distinct pasts.*

We denote the length of the interval order $\langle O, <\rangle$ underlying a history $h = \langle O, f, <\rangle$ as $\operatorname{len} h$.

Our history-weakening approximation functions A_k map histories to weaker histories whose corresponding interval orders have length at most k. While there are various ways to define such a function A_k, any such function enables the polynomial-time inclusion check $A_k(h) \in H(L)$ which we demonstrate in the following; Section 6.2 describes a polynomial-time computable instantiation of A_k which we have found useful in practice.

Interval orders have canonical representations which associate to operations integer-bounded intervals on the real number line; their canonical representations minimize the interval bounds to the interval-order length.

Lemma 6.2 (Greenough [9]). *An interval order* $\langle O, <\rangle$ *of length* $n \in \mathbb{N}$ *has a canonical representation* $I : O \to [n]^2$ *mapping each* $o \in O$ *to the interval* $I(o) = [i, j] \subseteq [0, n]$*, with*

$$i = |\{\mathrm{past}(o') : o' < o\}| \text{ and } j = |\{\mathrm{past}(o') : \neg o < o'\}| - 1$$

such that $o_1 < o_2$ *iff* $\sup I(o_1) < \inf I(o_2)$.

The canonical representation thus associates the interval $[i, j]$ to an operation o which succeeds the ith past, and precedes the $(j+1)$st past. Note that the interval of an operation can be determined in polynomial time by counting the distinct pasts among operations.

Example 6.2. *Figure 2(e) pictures the canonical representation of the history in Figure 2(b). Note that* $\operatorname{len} H(e) = 4$.

We define the *counting representation* $\Pi(h)$ of a history $h = \langle O, f, <\rangle$, whose underlying interval order $\langle O, <\rangle$ has the canonical representation $I : O \to [n]^2$, as the multiset

$$\Pi(h) = \{\{\langle f(o), I(o)\rangle : o \in O\}\}$$

of label-and-interval pairs; defining $\Pi(L) = \{\Pi(h) : h \in H(L)\}$ yields a criterion equivalent to history inclusion based on counting representations (which follows from Lemmas 4.1 and 6.2).

Lemma 6.3. $H(e) \in H(L)$ *iff* $\Pi(H(e)) \in \Pi(L)$.

Example 6.3. *The counting representation of the history* h *in Figure 2(b) is the multiset*

$$\{\{\, \langle \mathsf{push}(1), [0,0]\rangle, \langle \mathsf{pop} \Rightarrow 1, [1,1]\rangle, \langle \mathsf{pop} \Rightarrow 3, [1,3]\rangle, \\ \langle \mathsf{push}(2), [2,2]\rangle, \langle \mathsf{push}(3), [3,3]\rangle, \langle \mathsf{pop} \Rightarrow \mathtt{Empty}, [4,4]\rangle \,\}\}.$$

This counting representation leads to an effective logical characterization of history sets using arithmetic operations, inequalities, and a counting function #. Formally, *operation counting logic* is the first-order logic whose syntax is listed in Figure 3. The # function is interpreted over a history $h = \langle O, f, <\rangle$ as

$$[\![\#(\ell, i, j)]\!]_h = |\{o \in O : f(o) = \ell \text{ and } I(o) = [i, j]\}|$$

where I is the canonical representation of $\langle O, <\rangle$.

We allow predicates P of arbitrary arity over operation labels, so long as they are evaluated in polynomial time. Furthermore, operation-label variables are quantified only over the operation labels which occur in the history over which a formula is evaluated. The satisfaction relation $\models$ for quantified formulæ is defined by:

$$h \models \exists x.\, F \quad \text{iff} \quad \exists o \in O.\, h \models F[x \leftarrow f(o)].$$

$$\begin{array}{rl} i, j \in \mathbb{N} & \text{integer constants} \\ \ell \in \mathbb{L} & \text{operation-label constants} \\ x : \mathbb{L} & \text{operation-label variables} \\ X ::= & \ell \mid x \\ T ::= & i \mid \#(X, i, j) \mid T + T \\ F ::= & T \le T \mid P(X, \ldots, X) \mid \neg F \mid F \wedge F \mid \exists x.\, F \end{array}$$

Figure 3. The syntax of Operation Counting Logic.

where $h = \langle O, f, <\rangle$. The *quantifier count* of an operation counting formula Ψ is the number of quantified variables in Ψ. An operation counting formula Ψ *represents a library* L *up to* k when $h \in H(L)$ iff $h \models \Psi$ for every history h of length at most k.

Lemma 6.4. *Checking if a history* h *satisfies an operation counting formula* Ψ *of fixed quantifier count is decidable in polynomial time.*

Proof. This follows from the facts that: (1) the canonical representation of h is polynomial-time computable (see Lemma 6.2), (2) functions and predicates are polynomial-time computable, and (3) quantifiers are only instantiated over labels occurring in h. The latter implies that quantifiers can be replaced by a disjunction over the labels occurring in $h = \langle O, f, <\rangle$:

$$h \models \exists x.\, F \quad \text{iff} \quad h \models \bigvee_{o \in O} F[x \leftarrow f(o)]. \quad \square$$

6.2 Monitoring Bounded-Interval-Length Histories

Though there are numerous ways to define a function A_k which approximates histories to weaker k-interval-length histories, the natural solution we consider here is an A_k which maintains the last k interval bounds precisely, abstracting all previous interval bounds with equality. Formally, given a history $h = \langle O, <, f\rangle$ such that $\operatorname{len} h = n$, and $k \in \mathbb{N}$, we define $A_k(h) = \langle O, <', f\rangle$ by

$$o_1 <' o_2 \text{ iff } o_1 < o_2 \text{ and } n - k < \inf I(o_2)$$

where I is the interval map of h. Intuitively, A_k remembers only the ordering between "recent" operations which have started after interval $n - k$. It follows by definition that $A_k(h) \preceq h$, since ordering constraints are only removed from $<$. Note that $A_k(h) = h$ if $\operatorname{len} h \le k$.

Lemma 6.5. *Let* I *and* I_k *be the interval maps of* h *and* $A_k(h)$*. For each* $o \in O$ *with* $I(o) = [i, j]$*:*

$$I_k(o) = [\max(i - n + k, 0), \max(j - n + k, 0)].$$

We reduce the online maintenance of k-interval-length histories to the maintenance of integer counters. As the first step in establishing the link between history and counter maintenance, we define the $\oplus$ operator: For a given history $h = \langle O, f, <\rangle$ let

$$h \oplus m(u)_o = \langle O \cup \{o\}, f[o \mapsto m(u) \Rightarrow \bot], <'\rangle$$
$$h \oplus \mathsf{ret}(v)_o = \langle O, f[o \mapsto m(u) \Rightarrow v], <\rangle \text{ s.t. } f(o) = m(u) \Rightarrow \bot$$

where $<'$ is the transitive closure of $< \cup \{\langle o', o\rangle : f(o') = \_(\_) \Rightarrow v \neq \bot\}$ relating all completed operations to o. As the following lemma shows, the $\oplus$ operator allows us to manipulate our k-approximations $A_k(H(e))$ directly, without having to maintain the precise history $H(e)$ of an execution e. Note that besides maintaining $A_k(H(e))$, this formulation requires us to compute $A_{k-1}(H(e))$ periodically as well.

Lemma 6.6. *Let* $e' = e \cdot m(u)_o$.

- $A_k(H(e')) = A_k(H(e)) \oplus m(u)_o$ *if* $\operatorname{len} H(e') = \operatorname{len} H(e)$.

Data: Interval length $k \in \mathbb{N}$
Data: Stream $e \in (C \cup R)^\omega$ of call/return actions
Result: counting representation $\Pi(A_k(H(e)))$
initially $n = 0$, $s = \emptyset$, $\pi = \emptyset$
switch *input action* **do**
  if *call action* $m(u)_o$ **then**
    if *previous action was return* **then**
      incr n
      if $n > k$ **then** $\pi \leftarrow \overleftarrow{\pi}$
    end
    $s(o) \leftarrow n$
    incr $\pi(m(u) \Rightarrow \bot, [\min(n,k), k])$
  end
  case *return action* $\mathsf{ret}(v)_o$
    $i = s(o) - \max(n-k, 0)$
    decr $\pi(m(u) \Rightarrow \bot, [i, k])$
    incr $\pi(m(u) \Rightarrow v, [i, \min(n,k)])$
  end
endsw
yield π

Algorithm 1: An online operation algorithm for computing the approximation $\Pi(A_k(H(e)))$ of a given history $H(e)$.

- $A_k(H(e')) = A_{k-1}(H(e)) \oplus m(u)_o$ *if* $\operatorname{len} H(e') > \operatorname{len} H(e)$.

Additionally,

- $A_k(H(e \cdot \mathsf{ret}(v)_o)) = A_k(H(e)) \oplus \mathsf{ret}(v)_o$.

The second step in establishing the link between history and counter maintenance is given by the following lemma, which implies that history extension, i.e., execution step, corresponds to either a single counter increment, or a single decrement and a single increment.

Lemma 6.7. *Let* $\operatorname{len} h' = k$. *If* $h' = h \oplus m(u)_o$ *then*

- $\Pi(h') = \Pi(h) \cup \{\langle m(u) \Rightarrow \bot, [k,k]\rangle\}$.

If $h' = h \oplus \mathsf{ret}(v)_0$ *then*

- $\Pi(h') = \Pi(h) \cup \{\langle m(u) \Rightarrow v, [i,k]\rangle\} \setminus \{\langle m(u) \Rightarrow \bot, [i,k]\rangle\}$,

where I *is the interval map of* h' *and* $I(o) = [i,k]$.

As noted before Lemma 6.6, besides incrementing/decrementing counters as operations begin and complete, we must occasionally compute $A_{k-1}(H(e))$ as well, thus merging the least-recent interval bounds, in order to maintain a bounded-length history approximation. We achieve this with an interval shifting operation $\overleftarrow{\pi}$ applied to a counting representation π. Formally, for all $\ell \in \mathbb{L}$, and $0 \le i \le j \le k$, if ℓ is completed then

$$\begin{aligned}\overleftarrow{\pi}(\ell,[i,j]) = {} & \eta(\pi(\ell,[i+1,j+1]), i < k \wedge j < k) \\ & + \eta(\pi(\ell,[i,j+1]), i = 0 \wedge j < k) \\ & + \eta(\pi(\ell,[i,j]), i = 0 \wedge j = 0),\end{aligned}$$

and if ℓ is pending then

$$\overleftarrow{\pi}(\ell,[i,j]) = \eta(\pi(\ell,[i+1,k]), i < k) + \eta(\pi(\ell,[i,k]), i = 0),$$

where the conditional function $\eta : \mathbb{N} \times \mathbb{B} \to \mathbb{N}$ is defined by

$$\eta(n,\varphi) = \begin{cases} n & \text{if } \varphi \text{ is valid} \\ 0 & \text{otherwise.}\end{cases}$$

The following lemma establishes the intended effect of $\overleftarrow{\pi}$.

Lemma 6.8. *If* $\operatorname{len} h = k$ *and* $\pi = \Pi(h)$, *then* $\overleftarrow{\pi} = \Pi(A_{k-1}(h))$.

Algorithm 1 computes the counting representation of a given history incrementally, according to the increment/decrement and shift operations of Lemmas 6.7 and 6.8. The variable n maintains the interval length of the execution history $H(e)$, while the variable s maintains the infimum $s(o)$ of the interval of each operation o in $H(e)$; $s(o)$ can be discarded once o completes. The corresponding infimum in $A_k(H(e))$ when o is invoked is given by $\min(n,k)$, as the interval bounds used in $A_k(H(e))$ may not exceed k. When o completes, it is possible that some number of shift operations have translated its interval in $A_k(H(e))$. As the number of performed shift operations is equal to $\min(n-k,0)$, o's infimum in $A_k(H(e))$ when completing is given by $s(o) - \min(n-k,0)$. It follows by Lemmas 6.7 and 6.8 that Algorithm 1 computes $A_k(H(e))$.

Theorem 3. *Algorithm 1 computes the history approximation* $\Pi(A_k(H(e)))$ *of an execution* e *in* $O(|\mathbb{L}| \cdot k^2 + \operatorname{width}(e))$ *space*[2] *and* $O(|e|)$ *time, where* $\operatorname{width}(e)$ *is the maximum number of concurrent operations in* e.

In the worst case $\operatorname{width}(e) = |e|$. In practice however one expects the number of threads, and thus concurrent operations, to be bounded, nullifying the effect of $|e|$ on asymptotic space complexity.

Corollary 1. *The approximate inclusion problem* $A_k(h) \in H(L)$ *is decidable in polynomial time, for fixed* k, *given an operation counting formula* Ψ *for* L *up to* k, *of fixed quantifier count.*

As an online monitor program P_k for history approximation, Algorithm 1 effectively reduces the approximate refinement checking problem between L_1 and L_2 to a safety verification problem on the composition $P_k \times L_1$: as P_k tracks the counting representation $\pi = \Pi(A_k(H(e)))$ of the current execution e's approximation, we must only verify whether π satisfies the operation-counting formula Ψ for L_2 up to k.

Corollary 2. *Given* Ψ *an operation counting formula for* L_2 *up to* k, *the approximate inclusion* $\exists h.\ A_k(h) \in H(L_1) \setminus H(L_2)$ *is equivalent to the safety verification problem*

$$(P_k \times L_1) \models \Box\Psi$$

where Ψ *is interpreted over* P_k*'s counting representation.*

7. Atomic Collections

Atomic collection objects including stacks and queues are among the most heavily investigated concurrent objects [19]. We demonstrate that our approximation is effective in uncovering refinement violations for these objects, in the sense that most violations can be uncovered with coarse approximations, i.e., $k \le 3$, depending on the data structure. We achieve this by exhibiting families of operation counting formulæ characterizing these structures with "cutoff" (or "small-model") properties: informally, if $n \in \mathbb{N}$ is a cutoff for Ψ, then for any history h violating Ψ, there exists a weaker history $h' \preceq h$ violating Ψ with $\operatorname{len} h' \le n$. While this h' may not correspond directly to $A_k(h)$, for $k \le n$, we can deduce there exists some h' whose k-approximation $A_k(h')$ violates Ψ.

Our results build off of previous characterizations of concurrent data structures [1, 10] into a small set of constituent properties. For ease of presentation, in the following we consider only completed histories, i.e., in which no operation is pending. This restriction comes without loss of generality for library implementations which ensure that each operation can complete when uninterrupted by others [10]. In any case, our formulæ can be extended to handle pending operations as well.

[2] The space complexity of Algorithm 1 is constant in the number of operations when the size of integers is fixed, as in the case of modern computer architectures; otherwise, the space complexity is logarithmic in the number of operations, i.e., in execution length.

$$\mathsf{total}(x,i,j) = \sum_{i \le i' \le j' \le j} \#(x,i',j')$$

$$\mathsf{before}_k(x,y) = \bigvee_{0 \le i < k} \left(\begin{array}{l} \mathsf{total}(x,0,i) > 0 \wedge \\ \mathsf{total}(y,0,i) = 0 \wedge \mathsf{total}(x,i{+}1,k) = 0 \end{array} \right)$$

$$\mathsf{match}(x,y) = \mathsf{Push}(x) \wedge \mathsf{Pop}(y) \wedge \mathsf{SameVal}(x,y)$$

$$\Psi_{\mathrm{rv}} = \exists x, y.\ \mathsf{match}(x,y) \wedge \mathsf{before}_k(y,x)$$

$$\Psi_{\mathrm{ev}} = \exists x, y, z.\ \mathsf{match}(x,z) \wedge \mathsf{EmptyVal}(y) \wedge \mathsf{before}_k(x,y) \wedge \mathsf{before}_k(y,z)$$

$$\Psi_{\mathrm{fv}} = \exists x_1, x_2, y_1, y_2.\ \mathsf{match}(x_1,y_1) \wedge \mathsf{match}(x_2,y_2) \wedge \mathsf{before}_k(x_1,x_2) \wedge \mathsf{before}_k(y_2,y_1)$$

$$\Psi_{\mathrm{lv}} = \exists x_1, x_2, y_1, y_2.\ \mathsf{match}(x_1,y_1) \wedge \mathsf{match}(x_2,y_2) \wedge \mathsf{before}_k(x_1,x_2) \wedge \mathsf{before}_k(y_1,y_2) \wedge \mathsf{before}_k(x_2,y_1)$$

Figure 4. Four families of operation-counting formulæ characterizing atomic data structure violations, parameterized by the interval length $k \in \mathbb{N}$. The predicates $\mathsf{Push}(x)$, $\mathsf{Pop}(x)$, $\mathsf{EmptyVal}(x)$ hold when x is the label of a push, pop, or empty-pop operation, respectively, and $\mathsf{SameVal}(x,y)$ holds when x and y are labels with the same value, either in the argument or return position. All formulæ are of size polynomial in k, and have fixed quantifier counts.

7.1 Small-Model Formulæ for Stacks and Queues

A *formula family* Ψ is an indexed set $\{\Psi_i : i \in \mathbb{N}\}$. We say a history h *satisfies* Ψ, written $h \models \Psi$, when $h \models \Psi_i$ for $i = \operatorname{len} h$. We say that a family Ψ has a *cutoff* $n \in \mathbb{N}$ when for each history $h \models \Psi$ there exists some $h' \preceq h$ such that $A_n(h') \models \Psi$. Figure 4 defines four families of operation-counting formulæ:

- Ψ_{rv} characterizes *remove violations* in which a pop operation returns a value for which there is no corresponding push.
- Ψ_{ev} characterizes *empty violations* in which some pop operation returns EMPTY, yet whose span is covered by the presence of some pushed element which has not (yet) been popped.
- Ψ_{fv} characterizes *FIFO violations* in which some pair of pops occur in the *opposite* order of their corresponding pushes.
- Ψ_{lv} characterizes *LIFO violations* in which some pair of pops occur in the *same* order as their corresponding pushes, and the second push occurs before the first pop.

As in previous work [1, 10], our arguments for the (partial) completeness of these properties rely on *data independence* [28], i.e., that library executions are closed under consistent renaming $\mathbb{V} \to \mathbb{V}$ of method call/return values, and assume that each value is pushed at most once. In practice, collection-based data structures are data independent, and the second condition can always be met by tagging each value with a unique identifier. However, in order to achieve bounded counting representations, we may only distinguish between these values up to equivalence relations with finite quotients. Formally, we say a history $\langle O, f, < \rangle$ uses *unique values* when $f(o_1) = m_1(u) \Rightarrow v_1$ and $f(o_2) = m_2(u) \Rightarrow v_2$ implies $o_1 = o_2$.

Theorem 4. *The families Ψ_{rv}, Ψ_{ev}, Ψ_{fv}, and Ψ_{lv}, of operation-counting formulæ have cutoffs 0, 2, 2, and 3, respectively.*

Proof. Here we prove the theorem for Ψ_{rv} and Ψ_{fv}; the others follow similarly. Since Ψ_{rv} does not discriminate between intervals, $h = \langle O, <, f \rangle \models \Psi_{\mathrm{rv}}$ iff $A_0(h) = \langle O, \emptyset, f \rangle \models \Psi_{\mathrm{rv}}$.

Next, let $h = \langle O, <, f \rangle \models \Psi_{\mathrm{fv}}$ and let ℓ_1, ℓ_2, ℓ_1', and ℓ_2' be the instances of the variables x_1, x_2, y_1, and y_2, respectively. Let $\Psi_{\mathrm{fv},k}$ denote the formula parametrized by k in the family Ψ_{fv}. By definition $h \models \Psi_{\mathrm{fv},k}$, where $k = \operatorname{len} h$. We consider only the case when $\mathsf{total}(y_1) > 0$ holds (the other case is similar). Besides the constraints on operation labels, the formula $\Psi_{\mathrm{fv},k}$ states that all ℓ_1 operations end before an ℓ_1' operation starts and all ℓ_2 operations end before an ℓ_2' operation starts. Let $I : O \to [n]^2$ be the canonical representation of h. Also, let j_1 (resp., j_2) be the maximum upper bound of an interval associated to an ℓ_1 (resp., ℓ_3) operation, i.e.,

$$j_1 = \max\{j : \exists o \in O.\ I(o) = [i,j] \wedge f(o) = \ell_1\},$$
$$j_2 = \max\{j : \exists o \in O.\ I(o) = [i,j] \wedge f(o) = \ell_2\}.$$

We define a weaker history $h' \preceq h$, that contains exactly the same set of operations as h but it preserves only the ordering constraints $o_1 < o_2$ s.t. $I(o_1) \subseteq [0, j_1]$ and $I(o_2) \subseteq [j_1, \infty]$, or $I(o_1) \subseteq [0, j_2]$ and $I(o_2) \subseteq [j_2, \infty]$. The length of h' is at most 2 and $h' \models \Psi_{\mathrm{fv},2}$. Since $A_2(h') = h'$, Ψ_{rv} has cutoff 2. □

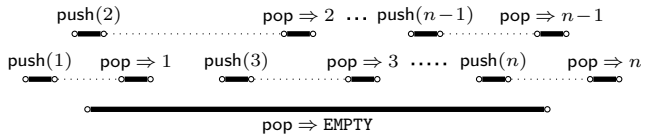


Figure 5. A family of empty violations, parameterized by $n \in \mathbb{N}$.

Corollary 3. *The families of operation-counting formulæ,*

$$\Psi_{\mathrm{stack}} = \Psi_{\mathrm{rv}} \vee \Psi_{\mathrm{ev}} \vee \Psi_{\mathrm{lv}}, \text{ and}$$
$$\Psi_{\mathrm{queue}} = \Psi_{\mathrm{rv}} \vee \Psi_{\mathrm{ev}} \vee \Psi_{\mathrm{fv}}$$

have cutoffs 3 and 2, respectively.

While Ψ_{rv} and Ψ_{fv} are complete, in the sense that they characterize every possible remove and FIFO violation, the Ψ_{lv} and Ψ_{ev} families are incomplete: Example 7.1 demonstrates this for Ψ_{ev} by exhibiting a parameterized history $h(n)$ for $n \geq 1$ such that $h(n) \notin H(L_{\mathrm{queue}})$,[3] yet $h(n) \models \neg\Psi_{\mathrm{ev}}$ for all $n > 1$. While it is theoretically possible to define a family $\Psi'_{\mathrm{ev},k}$ which catches all empty violations in histories with interval length at most k — a fact that we demonstrate in Section 7.2 — we do not currently know whether it is possible to construct a formula which catches all empty violations in histories with arbitrarily-large interval length.

Example 7.1. *In the history of Figure 5, n pairs of* push *and* pop *operations ensure that throughout the span of the* pop(EMPTY)*, some element is always present; i.e., at every time between the call and return of* pop(EMPTY)*, there exists some element $v \in \mathbb{V}$ such that* push(v) *has completed, yet* pop $\Rightarrow v$ *has not yet begun. It follows that any such history would not be included in the histories $H(L_{\mathrm{queue}})$ of an atomic queue library. The empty-violation family Ψ_{ev} however only captures such violations for $n = 1$, i.e., with a single* push-pop *pair spanning* pop(EMPTY).

Despite the theoretical possibility for empty violations which only surface for large n, our practical experience suggests small n tends to suffice. The bug B_5 of our static analysis experiment in Section 9.3 surfaced as an empty violation, and was detected with approximation A_1, implying $n \leq 1$, i.e., where some unmatched push operation precedes pop(EMPTY). Similarly, although the history of Figure 2(b) satisfies the empty-violation formula Ψ_{ev}, so does the weaker length-1 history of Figure 2(c). Furthermore, we have found that the same bugs which manifest as empty violations often manifest as order violations as well. The ABA bug of Figure 2(a) is such a case: had the first thread executed another push(4) before push(1), then the final pop of Thread 2 would have returned 4, yielding the order violation push(4); push(2); pop ⇒ 4.

[3] L_{queue} denotes an atomic FIFO queue library with push and pop methods.

7.2 Completeness for Bounded Interval Length

While Section 7.1 demonstrates complete families Ψ_{queue} and Ψ_{stack} of operation counting formulæ for the L_{queue} and L_{stack} objects for histories of interval length up to 2 and 3, respectively, this section exhibits complete formulæ for histories of any bounded interval length $k \in \mathbb{N}$.

For the rest of this section, we fix an interval length $k \in \mathbb{N}$ and demonstrate that a bounded enumeration of histories suffices to characterize every possible violation.

Given a history $h = \langle O, <, f\rangle$, an *element* $\langle o_1, o_2\rangle \in O^2$ is a pair of operations such that $f(o_1) = \mathsf{push}(v)$ and $f(o_2) = \mathsf{pop}{\Rightarrow}v$ for some $v \in \mathbb{V}$. We say that two distinct elements $\langle o_1, o_2\rangle$ and $\langle o_3, o_4\rangle$ are *redundant with* h when their operations have the same pasts and futures,[4]

$$\text{past}(o_1) = \text{past}(o_3) \qquad \text{future}(o_1) = \text{future}(o_3)$$
$$\text{past}(o_2) = \text{past}(o_4) \qquad \text{future}(o_2) = \text{future}(o_4)$$

(and thus will share the same canonical intervals). We say the element $\langle o_1, o_2\rangle$ is *redundant with* h when there exists an element of h with which it is redundant. We say that $h' = \langle O \cup O', <', f'\rangle$ *extends* $h = \langle O, <, f\rangle$ *with* O' when $o_1 < o_2$ iff $o_1 <' o_2$ for all $o_1, o_2 \in O$, and $f(o) = f'(o)$ for all $o \in O$.

Lemma 7.1. *Let $x \in \{\text{stack}, \text{queue}\}$. If h' extends h with $\{o_1, o_2\}$ and $\langle o_1, o_2\rangle$ is redundant with h, then $h \in H(L_x)$ iff $h' \in H(L_x)$.*

To enumerate the non-redundant k-length histories, let $\mathbb{I}_k = \{[i, j] : i, j \in \mathbb{N},\ 0 \le i \le j \le k\}$ be the set of integral intervals up to $k \in \mathbb{N}$, and fix an arbitrary total order $\ll$ over $\mathbb{I}_k^2$. We define the function H^1 mapping any subset $\{\langle I_{1,1}, I_{1,2}\rangle, \ldots, \langle I_{n,1}, I_{n,2}\rangle\} \subseteq \mathbb{I}_k^2$ to the history $\langle O, <, f\rangle$ where

$$O = \{\langle i, 1\rangle, \langle i, 2\rangle : 1 \le i \le n\}$$
$$\langle i_1, j_1\rangle < \langle i_2, j_2\rangle \text{ iff } I_{i_1,j_1} < I_{i_2,j_2}$$
$$f(\langle i, j\rangle) = \begin{cases} \mathsf{push}(i) & \text{if } j = 1 \\ \mathsf{pop} \Rightarrow i & \text{if } j = 2 \end{cases}$$

such that $\langle I_{1,1}, I_{1,2}\rangle \ll \langle I_{2,1}, I_{2,2}\rangle \ll \ldots \ll \langle I_{n,1}, I_{n,2}\rangle$. Then we define the finite set $H_k^1 = \{H^1(X) : X \subseteq \mathbb{I}_k^2\}$. Notice that the histories of H_k^1 do not contain redundant elements. Finally, we define the finite subset $H_{x,k}^1 = H_k^1 \setminus H(L_x)$ of histories which are not admitted by L_x, for $x \in \{\text{stack}, \text{queue}\}$, and enumerate them to write the formula characterizing length k violations to L_x:

$$\Psi_{x,k} = \bigvee \{\Psi_h : h \in H_{x,k}^1\}$$

where Ψ_h characterizes the history $h = \langle O, <, f\rangle$ with operations $\{o_1, \ldots, o_n\}$ and elements ξ, and its extensions:

$$\Psi_h = \exists x_1, \ldots, x_n. \bigwedge_{o_i < o_j} \mathsf{before}_k(x_i, x_j) \wedge \bigwedge_{\langle o_i, o_j\rangle \in \xi} \mathsf{match}(x_i, x_j)$$

Note that $\Psi_{x,k}$ is polynomial in size, and has fixed quantifier count, for fixed $k \in \mathbb{N}$.

Theorem 5. *Let $x \in \{stack, queue\}$. If $\text{len}\, h \le k$ and h uses unique values, then $h \in H(L_x)$ iff $h \models \neg\Psi_{x,k}$.*

8. Atomic Libraries with Context-Free Kernels

While the previous section provides operation-counting formulae for particular classes of atomic objects, here we provide a systematic technique to derive operation-counting formulae for any atomic library that can be written as a context-free language, including objects such as (reader-writer) locks and semaphores, or context-free approximations of arbitrary libraries. For this construction we require finite sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values. Given an atomic library L such that $\ker E(L)$ is context free, and an interval bound $k \in \mathbb{N}$, we generate a formula $\Psi_{L,k}$ representing L up to k.

Our construction relies on Parikh's Theorem [22]. We recall that the *Parikh image* of $w \in \Sigma^*$ is the multiset $\Pi(w) : \Sigma \to \mathbb{N}$ mapping each symbol $a \in \Sigma$ to its number of occurrences in w, and the *Parikh image* of a language $E \subseteq \Sigma^*$ is the set $\Pi(E) = \{\Pi(w) : w \in L\}$ of its words' images. If $\ker E(L)$ is context free, then the language $E_k = \{e \in E(L) : \text{len}\, H(e) \le k\}$ of L's executions with bounded-length histories has the same Parikh image as a context-free language, and by Parikh's Theorem, $\Pi(E_k)$ can be represented as a Presburger formula, from which we derive $\Psi_{L,k}$.

Theorem 6. *Let L be an atomic library over finite sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values such that $\ker E(L)$ is a context-free language, and let $k \in \mathbb{N}$. Then there exists an effectively-computable operation-counting formula $\Psi_{L,k}$ representing L up to k.*

This construction is useful in practice, allowing us to derive the formulæ used in our static-checking experiments of Section 9.3.

9. Experimental Evaluation

To demonstrate the practical value of our approach to refinement checking, we argue that our k-approximation:

- uncovers violations with small values of k,
- can be efficiently implemented for use in systematic concurrency testing and long-term runtime monitoring, and
- can be efficiently implemented for use in static analysis.

To argue these points, we have studied actual concurrent data structure implementations in C/C++, including the Scal[5] High-Performance Multicore-Scalable Computing suite. Some of these implementations, such as the Michael-Scott Queue [18], are meant to preserve observational refinement[6], while others, such as Kirsch et al.'s k-FIFO [12], are meant to preserve weaker properties. Unless otherwise noted, we use these implementations without modification, except to annotate methods with a fixed set of possible preemption points, e.g., preceding shared-memory accesses. While space prohibits including the entirety of our study, the sample and analysis which we do include is representative.

For our first two experiments (§9.1, §9.2) we have developed a tool for systematically enumerating a large number of alternate executions involving a limited number of object method invocations. We run each operation on a separate thread, and execute all thread schedules up to a given number $n \in \mathbb{N}$ of thread preemptions, at specified preemption points, similarly to Microsoft's Chess tool [20]. With $n = 0$ preemptions, there is only one schedule to execute, though the number of schedules grows exponentially as n increases. For instance, with our annotation of preemptions in Scal's Michael-Scott Queue, we execute $f(n)$ schedules of a program with 8 method invocations at a rate of roughly one million schedules per minute, where f is given by

$$f(1) = 33, f(2) = 612, f(3) = 8343, f(4) = 95434, f(5) = 930141$$

While similar in spirit to the second, our third experiment (§9.3) executes all round-robin thread schedules up to a given number $n \in \mathbb{N}$ of rounds *symbolically*: we use CSeq [7] to sequentialize a simple program which invokes a limited number of library methods, and then CBMC [13] (version 4.5) to perform bounded model checking up to a given loop-unroll bound. All measurements were made on similar MacBook Pro 2.XGHz Intel Core i5/i7 machines.

[4] Similarly to the definition of past from Section 6.1, we define the *future* of an operation o in an history $\langle O, <, f\rangle$ by $\text{future}(o) = \{o' \in O : o < o'\}$.

[5] `http://scal.cs.uni-salzburg.at`

[6] More precisely, they have been designed to be linearizable.

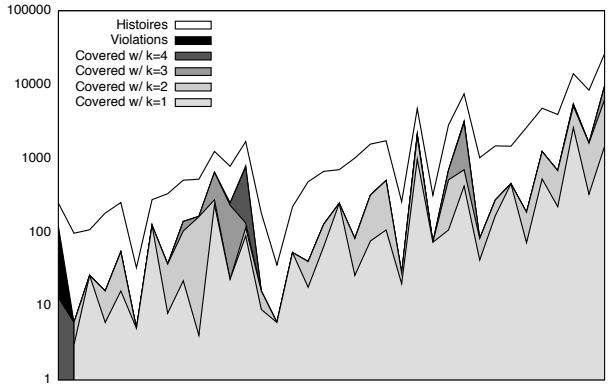


Figure 6. Comparison of violations covered with $k \leq 4$. Each data point counts histories on a logarithmic scale over all executions up to 5 preemptions on Scal's nonblocking bounded-reordering queue with $i \leq 4$ enqueue operations and $j \leq 4$ dequeue operations. The x-axis is ordered by increasing number of executions (1023–2359292) over $i+j$; we show only points with over 1000 executions. The largest data points measure the total number of unique histories encountered over a given set of executions. Second are the number of those histories violating refinement. Following are the numbers of those violations covered by A_k, for varying values of k. In this experiment A_0 exposed no violations.

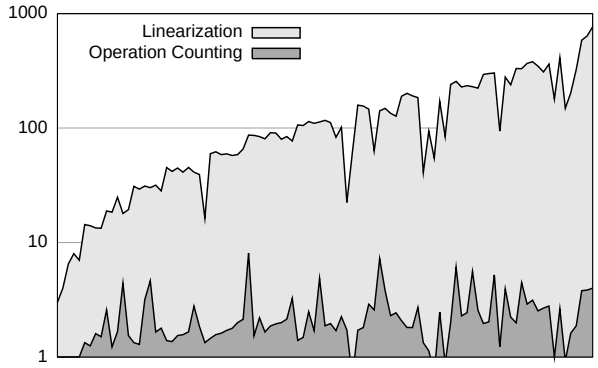


Figure 7. Comparison of runtime overhead between linearization-based monitoring and operation counting (for A_2) for up to 20 operations. Each data point measures runtime on a **logarithmic scale**, normalized to unmonitored execution time, over all executions up to 3 preemptions on Scal's nonblocking Michael-Scott queue with $i \leq 10$ enqueue operations and $j \leq 10$ dequeue operations. The x-axis is ordered by increasing $i+j$, and each data point is sampled from up to 126600 executions. Times do not include pre-calculation of sequential histories for linearization-based monitoring. While our A_k monitor scales well, usually maintaining under 3x overhead, the linearization monitor scales **exponentially**, running with nearly **1000x overhead** with 20 operations.

9.1 Coverage of Refinement Violations

To show that our approximation A_k uncovers observational refinement violations with small values of k, we measure the number of history violations detected by a traditional linearizability checker[7] versus those caught by A_k. The linearizability checker serves as an exact measure due to the equivalence of Section 5. While the approximation $A_k(H(e))$ of a violation $H(e)$ may not itself be a violation, one hopes to encounter some execution e' for which the violation $A_k(H(e'))$ *covers* $H(e)$, i.e., for which $A_k(H(e')) \preceq H(e)$.

Figure 6 demonstrates that A_k covers most or all violations with small values of k. While A_4 suffices to cover all violations at nearly all data points — besides the first point, where the sample size of 1023 executions is small relative to the 8 operations — all values $k > 0$ capture a nontrivial and increasing number of violations. Note that our cutoff results of Section 7 are not undermined by the fact that A_3 and A_4 have missed violations; violations were missed due to not having encountered a diverse enough sample of executions. In fact, as the execution-sample size increases (the x-axis is ordered by number of executions over operations) the value of k required to capture a violation appears to decrease, all violations being captured by A_3 after a certain point.

9.2 Operation Counting for Testing & Runtime Monitoring

Figure 7 compares the runtime overhead of our A_2 approximation versus a traditional linearizability checker[8] sampling executions with up to 20 operations on Scal's nonblocking Michael-Scott queue. Since computing the set $\ker H(L_{\text{queue}})$ of sequential queue histories over n operations becomes prohibitively expensive as n increases, surpassing a timeout of 5m for $n = 7$, we bypass the computation of $\ker H(L_{\text{queue}})$ entirely, simply enumerating the linearizations of a given execution history without checking inclusion. Despite our best-effort implementation, one observes the cost incurred by linearization: as the number of operations increases, the number of linearizations increases exponentially, and performance plummets. With only 20 operations, instrumentation overhead is nearly 1000x. Our counting-based implementation of A_2 avoids this dramatic overhead: though not seriously optimized, we observe a 2.01x geometric-mean runtime overhead. This scalability suggests that our approximation can be used not only for systematic concurrency testing with few method invocations, but also for runtime monitoring, where the number of operations grows without limit.

9.3 Operation Counting for Static Analysis

Our approximation A_k also leads to an effective static means of refinement checking, due to the simplicity of operation-counting. By using only simple increment and decrement operations on integer counters, and program assertions[9] on those counters, we are able to leverage static verification tools capable of integer reasoning. We have found that the major obstacle in applying static tools is concurrency: even though the required counter reasoning is simple, reasoning precisely about thread interleavings is challenging, with or without our operation-counting instrumentation.

Despite the difficultly of precise static reasoning about thread interleavings, we have successfully applied our approach using two different static-verification backends: one based on SMACK [24] and Corral [15], and the other based on CSeq [7] and CBMC [13]. Both toolchains are based on sequentialization [14] and SAT/SMT-based bounded model checking. Our results in Table 1 report on the latter. Among 5 data structure implementations, we manually injected 9 realistic concurrency bugs. All bugs were uncovered as refinement violations with approximation A_0 or A_1, in round-robin executions of up to 4 rounds, of a program with at most 4 push and 4 pop operations, and with loops unrolled up to 2 times. Although the time complexity of concurrent exploration is high independently of our operation-counting instrumentation, particularly as the number of rounds increases, one clearly observes that our approximation is effective in detecting violations statically.

[7] We implement a linearization-enumerating monitor, as in Line-Up [5].

[8] See footnote 7.

[9] We constructed program assertions automatically via Theorem 6.

Library	Bug	P	k	m	n	Time
Michael-Scott Queue	B_1 (`head`)	2×2	1	2	2	24.76s
Michael-Scott Queue	B_1 (`tail`)	3×1	1	2	3	45.44s
Treiber Stack	B_2	3×4	1	1	2	52.59s
Treiber Stack	B_3 (`push`)	2×2	1	1	2	24.46s
Treiber Stack	B_3 (`pop`)	2×2	1	1	2	15.16s
Elimination Stack	B_4	4×1	0	1	4	317.79s
Elimination Stack	B_5	3×1	1	1	4	222.04s
Elimination Stack	B_2	3×4	0	1	2	434.84s
Lock-coupling Set	B_6	1×2	0	2	2	11.27s
LFDS Queue	B_7	2×2	1	1	2	77.00s

Table 1. Runtimes for the static detection of injected refinement violations with CSeq & CBMC. For a given program $P_{i \times j}$ with i and j invocations to the `push` and `pop` methods, we explore the n-round round-robin schedules of $P_{i \times j}$ with m loop iterations unrolled, with a monitor for our A_k approximation. Bugs are (B_1) non-atomic lock operation, (B_2) ABA bug [17], (B_3) non-atomic CAS operation, (B_4) misplaced brace, (B_5) forgotten assignment, (B_6) misplaced lock, (B_7) predetermined capacity exceeded.

10. Related Work

Previous work on automated verification for concurrent objects focuses on the *linearizability* criterion [11]. While folklore has long held that linearizability implies observational refinement, Filipovic et al. [6] recently proved this fact; they also proved that the two criteria coincide when considering only library executions in which all operations have completed. Our results in Section 5 expound further: we show that the two criteria coincide for atomic object specifications, and that in general, observational refinement can hold for non-linearizable objects.

The theoretical limits of linearizability are well studied. Gibbons and Korach [8] show NP-completeness for checking a single execution. Alur et al. [2] show EXPSPACE membership for checking finite-state implementations against atomic specifications, but only when the number of program threads is bounded. Bouajjani et al. [4] show the same problem is undecidable with unbounded threads, and introduce a decidable variant for a restricted class of executions.

Several semi-automated approaches for proving linearizability, and thus observational refinement, have relied on annotating method bodies with *linearization points* [3, 16, 21, 27, 29], to reduce the otherwise-exponential space of possible history linearizations to one single linearization. These methods often rely on programmer annotation, and do not admit conclusive evidence of a violation in the case of a failed proof.

Automated approaches for detecting linearizability violations such as Line-Up [5] enumerate the exponentially-many possible history linearizations. This exponential cost effectively limits such approaches to executions with few operations, as noted in Section 9.2. Colt [25]'s approach mitigates this cost with programmer-annotated linearization points, as in the previously-mentioned approaches, and ultimately suffers from the same problem: a failed proof only indicates incorrect annotation.

References

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

[2] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[3] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07*, volume 4590 of *LNCS*, pages 477–490, 2007.

[4] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.

[5] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *PLDI '10*, pages 330–340. ACM.

[6] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[7] B. Fischer, O. Inverso, and G. Parlato. CSeq: a concurrency pre-processor for sequential C verification tools. In *ASE '13*, pages 710–713. IEEE, 2013.

[8] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

[9] T. L. Greenough. *Representation and Enumeration of Interval Orders and Semiorders*. PhD thesis, Dartmouth College, 1976.

[10] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013.

[11] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[12] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *PaCT 13*, volume 7979 of *LNCS*, pages 208–223, 2013.

[13] D. Kroening and M. Tautschnig. CBMC - C bounded model checker - (competition contribution). In *TACAS '14*, volume 8413 of *LNCS*, pages 389–391, 2014.

[14] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1): 73–97, 2009.

[15] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV '12*, volume 7358 of *LNCS*, pages 427–443.

[16] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *FM '09*, volume 5850 of *LNCS*, pages 321–337.

[17] M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Res. Ctr., 2004.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–275. ACM, 1996.

[19] M. Moir and N. Shavit. Concurrent data structures. In D. Metha and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 47:14–30. Chapman and Hall/CRC Press, San Jose, CA, 2007.

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08*, pages 267–280. USENIX Association, 2008.

[21] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC '10*, pages 85–94. ACM.

[22] R. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

[23] I. Rabinovitch. The dimension of semiorders. *Journal of Combinatorial Theory, Series A*, 25(1):50 – 61, 1978. ISSN 0097-3165.

[24] Z. Rakamaric and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV '14*, volume 8559 of *LNCS*, pages 106–113. Springer, 2014.

[25] O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA '11*, pages 51–64. ACM, 2011.

[26] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.

[27] V. Vafeiadis. Automatically proving linearizability. In *CAV '10*, volume 6174 of *LNCS*, pages 450–464.

[28] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL '86*, pages 184–193. ACM Press, 1986.

[29] S. J. Zhang. Scalable automatic linearizability checking. In *ICSE '11*, pages 1185–1187. ACM.