Monitoring Refinement via Symbolic Reasoning

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. 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*. Precisely testing this refinement even within a single execution is intractable, limiting existing approaches to executions with very few object invocations.

We develop scalable and effective algorithms for detecting refinement violations. Our algorithms are founded on incremental, symbolic reasoning, and exploit foundational insights into the refinement-checking problem. Our approach is *sound*, in that we detect only actual violations, and scales far beyond existing violation-detection algorithms. Empirically, we find that our approach is *practically* complete, in that we detect the violations arising in actual executions.

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: it is undecidable even for finite-state implementations whose methods can be called concurrently by arbitrarily-many threads [4]. In fact, even checking the conformance of a single program execution using one library with respect to another is intractable [10]. In practice, fully-automated techniques for checking observational refinement have been limited to detecting violations in executions with very few library-method invocations.

In this work we develop a fully-automated and highly-scalable means of detecting violations to observational refinement by monitoring program executions. The key challenge we address is how to achieve scalability while maintaining precision/completeness. Detecting refinement violations may require observing large executions, with many *operations*, i.e., object-method invocations. However, the complexity of precise violation-checking is exponential in the number of operations. Essentially, this check amounts to considering every possible *linearization* of an execution's operations, which are only partially-ordered by their happens-before relation. Only if none of the linearizations represent a valid sequence of method invocations does the execution witness a violation.

Our approach is based on sound yet possibly-incomplete means for avoiding the practical scalability pitfalls, the most immediate pitfall being the explicit enumeration of possible linearizations. We discover that naturally-occurring concurrent objects, including atomic collections, locks, and semaphores, can be expressed symbolically, in a first-order language over their method names, argument/return values, and invocation-ordering constraints. Ultimately this allows us to reduce violation detection for single executions to satisfiability in propositional logic. Practically speaking, this allows us to exploit the highly-developed algorithms of modern symbolic reasoning engines in place of the explicit enumeration of linearizations. Furthermore, symbolic reasoning lends itself to *incrementality*: as the symbolic representation of each successive execution step differs monotonically, only by the addition of a new operation or return value, we can reuse all logical implications of previous steps. Conceptually, this avoids recomputing the set of possible linearizations from scratch after each execution step.

While exploiting symbolic reasoning engines makes sense practically, and is likely to be more efficient, the link to symbolic reasoning also reveals insights leading to more-drastic optimizations. In particular, we notice that in proving satisfiability, the solver must essentially build a model of some linearization of the given execution which represents a valid object method-invocation sequence. In doing so, the solver may need to make branching decisions in addition to logical deductions, or *unit propagation*, possibly backtracking later, about

- how pending operations should be completed/dropped, and
- which operations should be linearized before others.

However, from the perspective of detecting violations, it is always sound to forgo such costly branching/backtracking and, for instance, wait until the given pending operations are actually completed later in the execution to determine their return values. Though it is unclear whether such strategies might actually be complete in theory, we hypothesize that they are effective in practice, uncovering the violations which surface in the logs of actual executions.

PLDI'15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
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.

Though limiting symbolic reasoning to saturation, or unit propagation, does avoid the exponential cost in the number of operations, a truly useful runtime monitor for observational refinement ought to be *linear* in the number of operations, and incur only a constant space overhead; otherwise it will progressively retard program execution and/or eventually exhaust program memory. Achieving this complexity goal implies that the monitor cannot store all previously-executed operations. However, simply forgetting arbitrary operations is unsound, in the sense that we may conclude a violation when none actually occurred. For instance, if a monitor for an atomic queue object observes ordered enqueue(a) and enqueue(b) operations, and dequeue(b), having dropped dequeue(a), the monitor may detect a violation, thinking dequeue(b) should not precede dequeue(a). To resolve this issue, we develop a sound theory for the removal of *matched* operations, e.g., the enqueue operations together with the dequeue operations removing the same added elements. We hypothesize that such strategies are, too, effective in practice, in that they continue to catch practically-occurring violations.

Empirically we demonstrate that our violation-detection strategies are profoundly-more scalable than existing techniques. We also validate our aforementioned completeness hypotheses: that in practice, violations are caught despite the possibility for incompleteness.

To summarize, we make the following contributions:

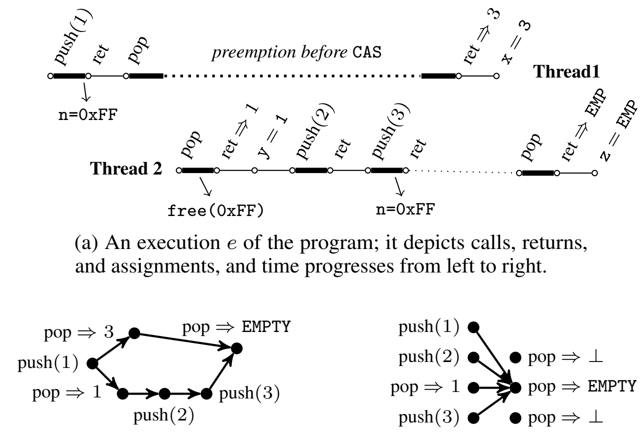
- First-order logical characterizations of naturally-occurring concurrent objects, allowing for symbolic reasoning about observational refinement (§3).
- A reduction from refinement-violation detection for single executions to propositional logic satisfiability (§4).
- A sound theory of matched-operation removal, allowing the scalability required for runtime monitoring (§5).
- Empirical validation that our violation-detection optimizations are scalable and effectively-complete (§6).

We begin by formalizing observational refinement (§2), and conclude with a discussion (§7) and mention of related work (§8).

2. Observational Refinement

Figure 1 lists a non-blocking stack [19] which stores its elements in a singly-linked list rooted at `Top`, and avoids blocking lock acquisitions in favor of non-blocking compare-and-swap (`CAS`) instructions in order to maximize parallelism. In one atomic step, the `CAS` instruction assigns `Top = n` only if `Top == t`.

Unfortunately this implementation suffers from a subtle concurrency bug [15] exposed by 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. This bug exposes the fact that our `CAS`-based implementation does not conform to programmers' expectations of a stack object whose operations execute atomically. In particular, the assignment `z = EMPTY` should never have occurred.



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

(b) The history of the above execution.

(c) A weaker history.

Figure 2. An execution and its history.

Formally this conformance is *observational refinement*. Essentially, a library implementation L_1 *refines* L_2 if every observable behavior of programs using L_1 is also observable using L_2. This is not the case between the `CAS`-based implementation L_1 of Figure 1 and an atomic implementation L_2, since `z = EMPTY` is observable with L_1 yet not with L_2.

We capture the interaction between programs and libraries by their *histories*, representing the partial happens-before orders of library method invocations. Fixing arbitrary sets $\mathbb{O}$, $\mathbb{M}$, and $\mathbb{V}$ of operation identifiers, method names, and parameter/return values, respectively, we define the set of *operation labels* as $\mathbb{L} = (\mathbb{M} \times \mathbb{V} \times (\mathbb{V} \cup \{\bot\}))$. We write $m(u) \Rightarrow v$ to denote the label $\ell = \langle m, u, v \rangle$. When $v = \bot$ we say ℓ is *pending*, and otherwise *completed*. A *history* $h = \langle O, <, f \rangle$ is a partial order $<$ on a set $O \subseteq \mathbb{O}$ of operations labeled by $f : O \to \mathbb{L}$ such that operations with pending labels are maximal. An operation o with label ℓ is an *ℓ-operation*, and o is *pending/completed* when ℓ is. We say h is *sequential* when $<$ is a total order on O, and *(in)complete* when (not) all operations are complete. A history set is *sequential* when it contains only sequential histories.

Example 2.1. *The history of Figure 2(b) captures the execution of Figure 2(a), where arrows depict the transitive reduction of the order relation. Essentially, operations o_1 and o_2 are ordered if o_1 happens before o_2. For example, although the push(1)-operation precedes pop $\Rightarrow 3$, since the push(1)-operation returns before the (pop $\Rightarrow 3$)-operation is called, the (pop $\Rightarrow 1$)-operation is incomparable to pop $\Rightarrow 3$, since it returns after the (pop $\Rightarrow 3$)-operation is called.*

This notion of histories gives rise to a natural *weaker-than* relation $\preceq$ relating any two histories h_1 and h_2 such that h_2 includes all completed operations of h_1, and preserves the order between the operations common with h_1. Pending operations in h_1 can be either omitted or completed in h_2. Formally, $\langle O_1, <_1, f_1 \rangle \preceq \langle O_2, <_2, f_2 \rangle$ iff there exists an injection $g : O_2 \to O_1$ such that

- $o \in \mathrm{ran}(g)$ 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 $v_1 \in \{v_2, \bot\}$. When the injection g need be fixed, we write $h_1 \preceq_g h_2$. We say h_1 and h_2 are *equivalent* when $h_1 \preceq h_2$ and $h_2 \preceq h_1$. 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. Finally, $\overline{H}$ denotes the closure $\{h : \exists h' \in H.\ h \preceq h'\}$ of a history set H under weakening.

Example 2.2. *The history of Figure 2(c) is weaker than that of Figure 2(b). While one of the two pending pop-operations is mapped to the completed (pop $\Rightarrow$ 3)-operation, the other is dropped.*

We model libraries as sets of histories. Since libraries only dictate methods' executions between their respective calls and returns, for any execution they admit, they must also admit executions with weaker inter-operation ordering, in which calls may happen earlier, and/or returns later. Thus any weakening of a history admitted by a library must also be admitted. Formally, a *library* L is a set of histories closed under weakening, i.e. $\overline{L} = L$, and a *kernel of* L is any set H such that $\overline{H} = L$. A library L is called *atomic* if it has a sequential kernel. Atomic libraries are often considered as specifications for concurrent objects. In practice, libraries can be made atomic by guarding their methods bodies with global lock acquisitions.

Example 2.3. *The* atomic stack *is the library whose unique kernel is the set of all sequential histories 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.*

We define *observational refinement* between two libraries as history-set inclusion, saying L_1 *refines* L_2 iff $L_1 \subseteq L_2$. Although this refinement is typically defined with respect to the admissibility of program executions, recent work shows these definitions are equivalent [5].

3. Refinement via Symbolic Reasoning

In this section we represent the kernels of typical concurrent objects, including atomic collections and locks, in a simple first-order language. Besides function and predicate symbols describing the operation labels and the order relation of a history, this language includes a predicate match describing a *matching function* M that maps operations o which remove or test the presence of values to the operations $M(o)$ which added them. For instance, a matching function M for a history of the atomic stack object is injective, and maps each (pop $\Rightarrow v_1$)-operation o to a push(v_2)-operation $M(o)$ such that $v_1 = v_2$ when $M(o)$ is defined. Similarly, the matching function M for an atomic lock object is also injective, and maps each unlock operation o to a lock operation $M(o)$ if $M(o)$ is defined.

Figures 3–7 list the properties characterizing typical concurrent objects in a first-order language whose variables range over operation identifiers, and whose functions and predicates are interpreted over the operation labels and order relation of a history, and a given matching function. We use the function symbols meth-od, arg-ument, and ret-urn, as well as the predicate symbols match and b-efore for this purpose. Note that we interpret the predicate $\mathsf{match}(x_1, x_2)$ by $o_1 = M(o_2)$ when each x_i binds to o_i. We represent the kernel H of each of the following objects by a first-order formula THEORY(H) such that $h \in H$ iff $h, M \models$ THEORY(H), for some M, where the satisfaction relation $\_, \_ \models \_$ is defined as usual, using the aforementioned interpretations.

ATOMIC

$\forall x.\ \neg \mathsf{b}(x,x) \wedge \forall x_1, x_2, x_3.\ (\mathsf{b}(x_1,x_2) \wedge \mathsf{b}(x_2,x_3)) \Rightarrow \mathsf{b}(x_1,x_3)$

$\forall x_1, x_2.\ \mathsf{b}(x_1,x_2) \oplus \mathsf{b}(x_2,x_1) \wedge \forall x.\ \mathsf{ret}(x) \neq \bot$

COMPLETED

$\forall x.\ \neg \mathsf{b}(x,x) \wedge \forall x_1, x_2, x_3.\ (\mathsf{b}(x_1,x_2) \wedge \mathsf{b}(x_2,x_3)) \Rightarrow \mathsf{b}(x_1,x_3)$

$\forall x.\ \mathsf{ret}(x) \neq \bot$

INJECTIVE

$\forall x, y_1, y_2.\ \mathsf{match}(x,y_1) \wedge \mathsf{match}(x,y_2) \Rightarrow y_1 = y_2$

INJECTIVE(X)

$\forall x, y_1, y_2.\ \mathsf{match}(x,y_1) \wedge \mathsf{match}(x,y_2) \wedge \mathsf{meth}(y_1) = \mathsf{meth}(y_2) = X$
$\quad \Rightarrow y_1 = y_2$

SYMMETRIC

$\forall x_1, x_2.\ \mathsf{match}(x_1,x_2) \Leftrightarrow \mathsf{match}(x_2,x_1)$

TOTAL(Y)

$\forall y.\ \mathsf{meth}(y) = Y \Rightarrow \exists x.\ \mathsf{match}(x,y) \wedge \mathsf{b}(x,y)$

MATCH1(X, Y)

$\forall x, y.\ \mathsf{match}(x,y) \Rightarrow \mathsf{meth}(x) = X \wedge \mathsf{meth}(y) = Y \wedge \mathsf{arg}(x) = \mathsf{ret}(y)$

MATCH2(X, Y_1, Y_2)

$\forall x, y.\ \mathsf{match}(x,y) \Rightarrow \mathsf{meth}(x) = X \wedge (\mathsf{meth}(y) = Y_1 \vee \mathsf{meth}(y) = Y_2)$
$\qquad \wedge\ \mathsf{arg}(x) = \mathsf{arg}(y)$

MATCH3(X, Y)

$\forall x, y.\ \mathsf{match}(x,y) \Rightarrow \mathsf{meth}(x) = X \wedge \mathsf{meth}(y) = Y$

Figure 3. Generic formulas used across many objects ($\oplus$ denotes exclusive or).

Example 3.1 (Atomic collections)**.** *The kernel of atomic queue objects is represented by the conjunction of properties stating:*

- *values are added before they are removed (*ADDREM*),*
- *values are removed in the order they are added (*FIFO*), and*
- *remove operations returning empty are not surrounded by matching adds and removes (*EMPTY*).*

We thus represent the kernel H_{q} of atomic queues by

$$\text{ATOMIC} \wedge \text{ADDREM} \wedge \text{FIFO} \wedge \text{EMPTY}$$

Similarly, we represent the kernel H_{pq} of priority queues by

$$\text{ATOMIC} \wedge \text{ADDREM} \wedge \text{MAX} \wedge \text{EMPTY}$$

and the kernel H_{st} of atomic stacks by

$$\text{ATOMIC} \wedge \text{ADDREM} \wedge \text{LIFO} \wedge \text{EMPTY}$$

Additionally, we enforce the sanity of the underlying matching function by adding the formulas MATCH1(add, rem) *and* INJECTIVE*, ensuring removes are matched to adds adding the removed value and that every two removes are matched to different adds.*

Example 3.2 (Atomic sets)**.** *Unlike the atomic queues and stacks which behave as* multisets *and return removed values, the atomic set's remove method takes as an argument a value to be removed, and succeeds whether or not the value is present. The formulas*

AddRem

$\forall r.\ \mathsf{meth}(r) = \mathsf{rem} \wedge \mathsf{ret}(r) \neq \mathsf{empty} \Rightarrow \exists a.\ \mathsf{match}(a, r) \wedge \mathsf{b}(a, r)$

Empty

$\forall e, a.\ \mathsf{meth}(e) = \mathsf{rem} \wedge \mathsf{ret}(e) = \mathsf{empty} \wedge \mathsf{meth}(a) = \mathsf{add}$
$\quad \wedge\ \mathsf{b}(a, e) \Rightarrow \exists r.\ \mathsf{match}(a, r) \wedge \mathsf{b}(r, e)$

Fifo

$\forall a_1, a_2, r_2.\ \mathsf{meth}(a_1) = \mathsf{add} \wedge \mathsf{match}(a_2, r_2)$
$\quad \wedge\ \mathsf{b}(a_1, a_2) \Rightarrow \exists r_1.\ \mathsf{match}(a_1, r_1) \wedge \mathsf{b}(r_1, r_2)$

Lifo

$\forall a_1, a_2, r_1.\ \mathsf{meth}(a_2) = \mathsf{add} \wedge \mathsf{match}(a_1, r_1)$
$\quad \wedge\ \mathsf{b}(a_1, a_2) \wedge \mathsf{b}(a_2, r_1) \Rightarrow \exists r_2.\ \mathsf{match}(a_2, r_2) \wedge \mathsf{b}(r_2, r_1)$

Max

$\forall a_1, a_2, r_1.\ \mathsf{meth}(a_2) = \mathsf{add} \wedge \mathsf{match}(a_1, r_1) \wedge \mathsf{b}(a_2, r_1)$
$\quad \wedge\ \mathsf{arg}(a_1) < \mathsf{arg}(a_2) \Rightarrow \exists r_2.\ \mathsf{match}(a_2, r_2) \wedge \mathsf{b}(r_2, r_1)$

Figure 4. Formulas for collection objects.

Include *and* Exclude *specify when a contains-operation may return true. We represent the kernel H_s of atomic sets by*

$$\text{Atomic} \wedge \text{Include} \wedge \text{Exclude}$$

Additionally, we enforce the sanity of the underlying matching function by adding the formulas: Match2(add, remove, contains) *ensuring removes and contains returning true are matched to adds of the same value,* Injective(remove) *ensuring that every two removes are matched to different adds,* AddRem *ensuring that the matching function maps removes to preceding adds, and* MatchingAdd *(resp.,* MatchingRem*) ensuring that every add matched to a remove (resp., remove matched to an add) is the first in a sequence of adds adding (resp., removes removing) the same value.*

Example 3.3 (Atomic register)**.** *Atomic registers with read and write methods essentially ensure that each value read is written by the most recent* write*-operation. We represent the kernel H_r of atomic registers by*

$$\text{Atomic} \wedge \text{ReadWrite} \wedge \text{ReadFrom}$$

and enforce the sanity of the underlying matching function by adding the formulas Injective *and* Match1(write, read)*, ensuring reads are matched to writes writing the read value.*

Example 3.4 (Synchronization objects)**.** *Atomic lock objects with lock and unlock methods ensure that at most one thread holds a lock at any moment. We represent the kernel of atomic locks by*

$$\text{Atomic} \wedge \text{Total}(\mathsf{unlock}) \wedge \text{Mutex}$$

and enforce coherent matching by adding Match3(lock, unlock) *and* Injective *formulas. Atomic semaphore objects with acquire and release methods generalize atomic locks, ensuring that at most n copies of a resource are held at any moment, for some fixed $n \in \mathbb{N}$. We represent the kernel of atomic semaphores by*

$$\text{Atomic} \wedge \text{Total}(\mathsf{release}) \wedge \text{Limit}$$

and enforce coherent matching by adding Match3(acquire, release) *and* Injective *formulas.*

Exchanger objects are used to pair up threads so they can atomically swap values. The only method of this object is exchange, which receives as input a value v that a thread it offers to swap and returns a value $v' \neq$ `null` if it has paired up with an exchange(v')

Include

$\forall c.\ \mathsf{meth}(c) = \mathsf{contains} \wedge \mathsf{ret}(c) = \mathsf{true}$
$\quad \Rightarrow \exists a.\ \mathsf{match}(a, c) \wedge \mathsf{b}(a, c)$

$\forall c, a, r.\ \mathsf{meth}(c) = \mathsf{contains} \wedge \mathsf{ret}(c) = \mathsf{true}$
$\quad \wedge\ \mathsf{meth}(r) = \mathsf{remove} \wedge \mathsf{meth}(a) = \mathsf{add} \wedge \mathsf{match}(a, c) \wedge \mathsf{match}(a, r)$
$\quad \Rightarrow \mathsf{b}(a, c) \wedge \mathsf{b}(c, r)$

Exclude

$\forall c, a.\ \mathsf{meth}(c) = \mathsf{contains} \wedge \mathsf{ret}(c) = \mathsf{false}$
$\quad \wedge\ \mathsf{meth}(a) = \mathsf{add} \wedge \mathsf{arg}(c) = \mathsf{arg}(a) \wedge \mathsf{b}(a, c)$
$\quad \Rightarrow \exists r.\ \mathsf{meth}(r) = \mathsf{remove} \wedge \mathsf{match}(a, r) \wedge \mathsf{b}(r, c)$

AddRem

$\forall a, r.\ \mathsf{match}(a, r) \wedge \mathsf{meth}(r) = \mathsf{remove} \Rightarrow \mathsf{meth}(a) = \mathsf{add} \wedge \mathsf{b}(a, r)$

MatchingAdd

$\forall a, x, p.\ \mathsf{match}(a, x) \wedge \mathsf{arg}(p) = \mathsf{arg}(a)$
$\quad \wedge\ \forall q.\ \mathsf{b}(q, p) \vee \mathsf{b}(a, q) \vee \mathsf{arg}(q) \neq \mathsf{arg}(a)$
$\quad \Rightarrow \mathsf{meth}(p) = \mathsf{remove} \vee (\mathsf{meth}(p) = \mathsf{contains} \wedge \mathsf{ret}(p) = \mathsf{false})$

MatchingRem

$\forall a, r, p.\ \mathsf{match}(a, r) \wedge \mathsf{arg}(p) = \mathsf{arg}(r)$
$\quad \wedge\ \forall q.\ \mathsf{b}(q, p) \vee \mathsf{b}(r, q) \vee \mathsf{arg}(q) \neq \mathsf{arg}(r)$
$\quad \Rightarrow \mathsf{meth}(p) = \mathsf{add} \vee (\mathsf{meth}(p) = \mathsf{contains} \wedge \mathsf{ret}(p) = \mathsf{true})$

Figure 5. Formulas for set objects.

ReadWrite

$\forall r.\ \mathsf{meth}(r) = \mathsf{read} \Rightarrow \exists w.\ \mathsf{match}(w, r) \wedge \mathsf{b}(w, r)$

ReadFrom

$\forall w_1, r.\ \mathsf{meth}(w_1) = \mathsf{write} \wedge \mathsf{meth}(r) = \mathsf{read} \wedge \neg\mathsf{match}(w_1, r)$
$\quad \wedge\ \mathsf{b}(w_1, r) \Rightarrow \exists w_2.\ \mathsf{match}(w_2, r) \wedge \mathsf{b}(w_1, w_2) \wedge \mathsf{b}(w_2, r)$

Figure 6. Formulas for register objects.

operation. The latter operation will return the value v. We represent the kernel of exchanger objects by

$$\text{Completed} \wedge \text{Exchange}$$

The kernel of this object contains non-sequential histories because the time spans of exchange operations that pair up overlap. Additionally, we enforce the sanity of the underlying matching function by adding the formulas: Match1(exchange, exchange)*,* Injective*, and* Symmetric *ensuring that the matching function is injective and symmetric and that it associates operations returning a value v to operations that receive v as input, and* MatchOverlap *ensuring that matched operations overlap in time.*

4. Refinement via Propositional Reasoning

In this section we demonstrate that the history membership problem $h \in \overline{H}$ reduces to propositional satisfiability, given a formula Theory(H) characterizing the histories of the library kernel H. Note that $h \in \overline{H}$ iff h is weaker than some history $h' \in H$, or equivalently weaker than some history h' such that $h', M \models$ Theory(H), for some matching function M. When h is complete, the fact that any stronger history contains exactly the same set of operations enables the construction of a formula Stronger(h) characterizing the histories stronger than h. Together

MUTEX

$\forall \ell_1, \ell_2.\ \mathsf{meth}(\ell_1) = \mathsf{meth}(\ell_2) = \mathsf{lock} \wedge \mathsf{b}(\ell_1, \ell_2)$
$\quad \Rightarrow \exists u.\ \mathsf{meth}(u) = \mathsf{unlock} \wedge \mathsf{b}(\ell_1, u) \wedge \mathsf{b}(u, \ell_2)$

LIMIT

$\forall x_0, \ldots, x_n.\ \bigwedge_{0 \le i < n} \mathsf{b}(x_i, x_n) \wedge \bigwedge_{0 \le i \le n} \mathsf{meth}(x_i) = \mathsf{acquire}$
$\quad \Rightarrow \exists r.\ \mathsf{b}(r, x_n) \wedge \bigvee_{0 \le i \le n} \mathsf{match}(r, x_i)$

EXCHANGE

$\forall x.\ \mathsf{ret}(x) \neq \mathsf{null} \Rightarrow \exists y.\ \mathsf{match}(x, y)$

MATCHOVERLAP

$\forall x_1, x_2.\ \mathsf{match}(x_1, x_2) \Rightarrow \neg \mathsf{b}(x_1, x_2) \wedge \neg \mathsf{b}(x_2, x_1)$

Figure 7. Formulas for synchronization objects.

with THEORY(H), this formula describes all stronger histories satisfying THEORY(H), and is therefore equivalent to $h \in \overline{H}$. We show how to construct these formulas in Section 4.1.

When h contains pending operations, stronger histories h' may contain fewer operations, since some pending operations of h may be omitted in h', and others completed. In this case the joint satisfiability of THEORY(H) and STRONGER(h) must be enhanced with additional constraints to ensure that the operations of h' include at least the completed operations of h, and possibly some pending operations of h. We tackle this problem in Section 4.2.

4.1 Complete Histories

The following lemma characterizes the weaker than relation between histories. It states that a history h' stronger than a complete history h can only differ in having more order constraints between the operations, the operation labels being the same in h and h'.

Lemma 4.1. *A complete history $h = \langle O, <, f \rangle$ is weaker than another history $h' = \langle O', <', f' \rangle$ iff there exists a bijection $g : O' \to O$ such that:*

- *operations related by g have the same label, i.e., for each $o \in O$, $f(o) = f'(g^{-1}(o))$, and*
- *order constraints are preserved from h to h', i.e., for each $o, o' \in O$, $o < o'$ implies $g^{-1}(o) <' g^{-1}(o')$.*

We characterize histories stronger than h by the formula STRONGER(h), defined as the conjunction

$$\text{DOMAIN}(h) \wedge \text{LABELS}(h) \wedge \text{ORDER}(h)$$

using the formulas of Figure 8 characterizing the identifiers, labels, and order constraints of h. Note that the DOMAIN(h) formula restricts the interpretation domain of each variable to the operations of h. As a consequence of Lemma 4.1, the formula STRONGER(h) does indeed characterize all histories at least as strong as h.

Lemma 4.2. *Let h and h' be histories. Then $h \preceq h'$ iff $h' \models$ STRONGER(h).*

It follows that the library membership test $h \in \overline{H}$ for complete histories h reduces to first-order satisfiability.

Theorem 1. *Let h be a complete history, and H a history set. Then $h \in \overline{H}$ iff* STRONGER$(h) \wedge$ THEORY(H) *is satisfiable.*

As long as the formula THEORY(H) contains only a fixed set of predicates e.g., $=$ and $\le$, as is the case for all formulas of Figures 3–7, satisfiability of STRONGER$(h) \wedge$ THEORY(H) reduces to propositional satisfiability. Intuitively this holds since the domain of (quantified) variables is restricted to operations appearing in h. Thus for a given h and H, we construct the propositional formula $[\![\text{STRONGER}(h) \wedge \text{THEORY}(H)]\!]$ by replacing each universally-quantified subformula $\forall x.\varphi$ by $\bigwedge_{o \in O} \varphi[x \mapsto o]$, and each existentially-quantified subformula $\exists x.\varphi$ by $\bigvee_{o \in O} \varphi[x \mapsto o]$. It follows that this propositional formula is equisatisfiable to the original first-order formula, and is constructed in polynomial time.

DOMAIN(h) $\quad \bigwedge_{o_1, o_2 \in O} o_1 \neq o_2 \wedge \forall x.\ \bigvee_{o \in O} x = o$

LABELS(h) $\quad \bigwedge_{f(o) = (m(u) \Rightarrow v)} \mathsf{meth}(o) = m \wedge \mathsf{arg}(o) = u \wedge [\mathsf{ret}(o) = v]_{v \neq \bot}$

ORDER(h) $\quad \bigwedge_{o_1 < o_2} \mathsf{b}(o_1, o_2)$

USED $\quad \forall x.\ \mathsf{ret}(x) \neq \bot \Rightarrow \mathsf{used}(x)$

Figure 8. Formulas characterizing histories $h = \langle O, <, f \rangle$ ($\mathsf{ret}(o) = v$ is present in LABELS(h) only if $v \neq \bot$).

Corollary 1. *Let h be a complete history, and H a history set. Then $h \in \overline{H}$ iff the propositional formula*

$$[\![\text{STRONGER}(h) \wedge \text{THEORY}(H)]\!]$$

is satisfiable.

4.2 Incomplete Histories

The pending operations of a history h may be omitted in a stronger history or they may be completed with arbitrary return values. Therefore, the set of histories stronger than a history h can be characterized by a formula obtained from STRONGER(h) by adding a domain predicate used that is constrained to contain all the completed operations of h and by omitting the constraints on the return values of pending operations. Moreover, every operation of h whose return value is different from $\bot$ in a model of this formula (this may be an operation which is pending in h) should satisfy used. It can be proved that every history stronger than h corresponds to a model of this formula, projected on the set of operations satisfying used.

Thus, we override the STRONGER(h) formula for incomplete histories h as the conjunction

$$\text{DOMAIN}(h) \wedge \text{LABELS}(h) \wedge \text{ORDER}(h) \wedge \text{USED},$$

where USED is defined in Figure 8.

For arbitrary, not necessarily complete, histories h, the models of STRONGER(h) are histories paired with an interpretation $U : O \to \mathbb{B}$ for the domain predicate used, mapping the operations O of h to $\{\mathsf{true}, \mathsf{false}\}$. Given such a model $\langle h, U \rangle$, let $U(h)$ be the history obtained from h by deleting operations o such that $\neg U(o)$.

Lemma 4.3. *Let h and h' be histories, and $U : O' \to \mathbb{B}$. Then $h', U \models$ STRONGER(h) iff $h \preceq U(h')$.*

We leverage the predicate used to guard the domain of quantifiers in THEORY(H). For simplicity, we assume that THEORY(H) is given in prenex normal form, whose quantifier prefix is of the form $\forall^*\exists^*$. All of the formulas of Figures 3–7 can be written in this form. We thus define the guarded formula $G(\varphi)$ of the formula $\varphi = \forall \vec{x}.\exists \vec{y}.\psi$ as

$$G(\varphi) = \forall \vec{x}.\exists \vec{y}.\ \bigwedge_{x_i} \mathsf{used}(x_i) \Rightarrow \bigwedge_{y_i} \mathsf{used}(y_i) \wedge \psi$$

As usual, universal quantifiers are guarded using implication and existential quantifiers using conjunction. It follows from Lemma 4.3 that history membership reduces to first-order satisfiability.

Theorem 2. *Let h be a history, and H a history set. Then $h \in \overline{H}$ iff the first-order formula*

$$\text{STRONGER}(h) \wedge G(\text{THEORY}(H))$$

is satisfiable.

We again reduce this first-order satisfiability problem to propositional satisfiability by limiting the domain of quantifiers to the operations of h via the function $[\![\cdot]\!]$.

Corollary 2. *Let h be a history, and H a history set. Then $h \in \overline{H}$ iff the propositional formula*

$$[\![\text{STRONGER}(h) \wedge G(\text{THEORY}(H))]\!]$$

is satisfiable.

5. Removing Matched Operations

While the reduction to symbolic reasoning enabled by the previous sections already offers practical advantages over the explicit enumeration of history linearizations, this reduction does nothing to avoid the increasing cost of refinement checking as execution-length increases. A truly useful runtime monitor must be *linear* in the number of operations in order to avoid a progressive slowdown of the monitored implementation. Achieving this complexity goal implies forgetting increasingly-many previously-executed operations. However, forgetting arbitrary operations from valid histories can result in falsely-reported violations. For example, removing the write(1) operation from the valid atomic register history of Figure 10 results in an invalid history. We leverage the same notion of operation matching used to characterize library kernels in order to identify groups of operations whose removal preserves history membership.

5.1 Unique matching functions

Recall that a history $h \in \overline{H}$ iff there exists a history h' stronger than h and a matching function $M : O \rightharpoonup O$ such that $h', M \models \text{THEORY}(H)$. We assume in the following that the matching function $M : O \rightharpoonup O$ of every history h is uniquely determined by the operation labels in h, i.e., there exists a function $\mathsf{M} : \mathbb{L} \rightharpoonup \mathbb{L}$ such that $M(o) = o'$ iff $\mathsf{M}(f(o)) = f(o')$. The matching function associated to some history h is denoted by M_h. We give some examples of how to construct histories of standard libraries with unique matching functions.

Example 5.1 (Collections)**.** *For usual implementations of collections such as stacks, queues, and sets, each operation adding a value to the collection is going to receive as input a value which is uniquely identified by a tag. When a method removing an element from the collection succeeds it is also going to return the unique tag associated with that element, thus defining a unique matching function from remove operations to add operations. The same strategy can be adapted to implementations of a register: the inputs to write operations are tagged and the read operations return tagged values.*

Example 5.2 (Locks)**.** *The implementations of a lock object usually have two abstract states, one where the object is unlocked, and one where it is locked. The lock operations can be modified to receive as input a value which is unique for every lock operation in an execution and every successful execution of a lock operation results in an object state that stores that input value. When an unlock operation succeeds, it returns the value stored in the object state. Therefore, the matching function maps (*unlock $\Rightarrow v$*) operations to* lock(v) *operations.*

Example 5.3 (Semaphores)**.** *Semaphore objects are usually implemented using a counter, which counts the number of acquire operations which successfully entered the semaphore and which are not yet released. We can instrument the implementation by keeping a map which maps each* slot *(from 1 to the capacity c) to a unique tag which was received as input by the acquire operation which has that slot, if any. When a release succeeds in decrementing the counter, it returns the unique tag of the acquire which had the slot.*

5.2 Closure under removing matched operations

A *match* of a history h is an operation o together with the maximal set of operations mapped by M_h to o. Moreover, a match consists only of completed operations and at least one operation mapped to o. Formally, a match of a history $h = \langle O, <, f \rangle$ is a set of operations $m = o \cup M_h^{-1}(o)$ such that $o \in O$, $M_h^{-1}(o) \neq \emptyset$, and all the operations in m are completed. The operation o of a match $m = o \cup M_h^{-1}(o)$ is denoted by $+(m)$.

Example 5.4. *Let h be the history in Figure 9 such that the matching function M_h maps every (*rem $\Rightarrow i$*) operation to the* add(i) *operation. The matches of h are* $m_1 = \{\text{add}(1), \text{rem} \Rightarrow 1\}$, $m_2 = \{\text{add}(2), \text{rem} \Rightarrow 2\}$, *and* $m_3 = \{\text{add}(3), \text{rem} \Rightarrow 3\}$. *Since every operation has a different label, we abuse the notation and write matches as sets of operation labels. Then,* $+(m_i) = \text{add}(i)$, *for each i.*

A set of histories H is *match-removal closed* iff for every history $h \in H$ and every match m of h, H contains the history obtained from h by deleting the operations in m. The history obtained from another history h by deleting a set of operations O is denoted by $h \setminus O$.

The kernels of all the reference implementations described in Section 3 are match-removal closed. For instance, consider a sequential history h in the basis of an atomic queue and a match $m = \{\text{add}(1), \text{rem} \Rightarrow 1\}$. The history obtained from h by removing the match m is also a valid sequential queue history because essentially, the remaining values are still removed in the order in which they are added. Match-removal closure follows from the fact that for every such kernel H, the formula $\text{THEORY}(H)$ holds for $h \setminus m$ whenever it holds for h, for any history h and m an arbitrary match of h.

Theorem 3. *The kernels of the atomic queue, stack, set, register, lock, semaphore, and exchanger are match-removal closed.*

The match-removal closure property extends from a kernel H to the entire library $\overline{H}$. Therefore, if by deleting matches from a history we get a history which is not admitted by a library $\overline{H}$ then the initial history is also not admitted by $\overline{H}$.

Theorem 4. *$\overline{H}$ is match-removal closed if H is.*

Proof. Let $h \in \overline{H}$ and m a match of h. By definition, there exists a history $h' \in H$ such that $h \preceq h'$. Since the matching functions are uniquely determined by the operation labels, m is also a match of h'. By hypothesis, H is match-removal closed which implies that the history h'' obtained from h' by deleting the operations in m is also in H. From the definition of $\preceq$ it follows that the history $h \setminus m$ is weaker than h'' which implies $h \setminus m \in \overline{H}$. □

While the statement of Theorem 4 implies that a history $h \setminus m$, where m is a match of h, belongs to $\overline{H}$ whenever $h \in \overline{H}$, Example 5.5 shows that the reverse is not true.

Example 5.5. *Figure 9 pictures a history h which is not admitted by the atomic stack: since the element 3 was pushed after 2, (*rem $\Rightarrow 3$*) should not have started after (*rem $\Rightarrow 2$*) has finished. The matching function M_h maps every (*rem $\Rightarrow i$*) operation to the* add(i) *operation.*

The history obtained by removing the match $\{\text{add}(2), \text{rem} \Rightarrow 2\}$ *is however admitted by the atomic stack.*

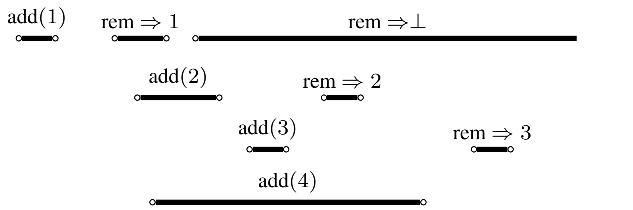


Figure 9. A history that is not admitted by the atomic stack. Each operation is represented by an horizontal line segment. The line segment of an operation ending before the line segment of another operation means that the two operations are ordered (reading from left to right).

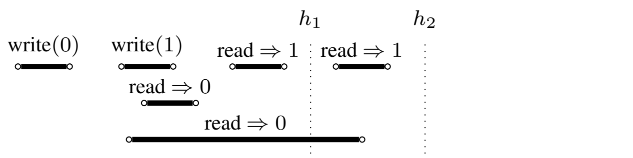


Figure 10. Two histories h_1 and h_2 of the atomic register, where h_2 is an extension of h_1.

5.3 Forgetting matched operations

A runtime monitor continuously checking history membership can soundly forget matches provided that Theorem 4 holds and that every match of a history h is also a match of every extension of h with new operations (a match of h could be strictly included in a match of an extension and therefore, not a match of the extension). If the latter doesn't hold, then removing an arbitrary match before extending a history may be the same as removing a set of operations which is not a match from the extended history (and Theorem 4 wouldn't apply). When the matching function of all the histories in the library is injective, every match has exactly two operations and this property trivially holds. All the reference implementations in Section 3 have injective matching functions except for the atomic sets and registers. Nevertheless, we prove in the following that the atomic sets and registers also satisfy this property (that every match of a history h is also a match of every extension of h).

A history h_2 *extends* h_1 iff h_2 is the history of an execution that extends the execution represented by h_1. Formally, $h_2 = \langle O_2, <_2, f_2\rangle$ extends $h_1 = \langle O_1, <_1, f_1\rangle$, written $h_1 \rhd h_2$, iff

- $O_1 \subseteq O_2$,
- $f_1(o) \ll f_2(o)$ for each $o \in O_1$,
- $o_1 <_1 o_2$ iff $o_1 <_2 o_2$ for each $o_1, o_2 \in O_1$, and
- $o_1 <_2 o_2$ for each $o_1 \in O_1$ and $o_2 \in O_2 \setminus O_1$.

Lemma 5.1. *Let $\overline{H}$ be a library such that M_h is injective for all $h \in \overline{H}$. For every two histories $h_1, h_2 \in \overline{H}$ such that $h_1 \rhd h_2$, if m is a match of h_1 then m is also a match of h_2.*

Example 5.6 shows that this result doesn't hold when the matching function is not injective.

Example 5.6. *Figure 10 pictures two histories h_1 and h_2 of the atomic register, the latter being an extension of the former. We assume that the matching function maps every (*read $\Rightarrow i$*) operation to the* write(i) *operation. The match* $\{\text{write}(1), \text{read} \Rightarrow 1\}$ *of h_1 is strictly included in the match* $\{\text{write}(1), \text{read} \Rightarrow 1, \text{read} \Rightarrow 1\}$ *of h_2, and therefore not a match of h_2. Removing the match of h_1 and extending it with the (*read $\Rightarrow 1$*) operation results in a spurious violation.*

For libraries with non-injective matching functions, we identify conditions on their kernels which imply that a match m of a history h is a match of every extension of h.

Let R be a relation on operation labels. We say that a history h is *R-ordered* iff for any two matches m_1 and m_2 of h such that $R(+(m_1), +(m_2))$, each operation of m_1 is ordered before each operation of m_2.

Consider the atomic register and the relation R_{reg} which holds for every two labels of two write operations. Then, any history from its kernel (which by definition is sequential) is R-ordered because every read operation is mapped by the matching function to the closest preceding write operation. Similarly, one can show that any history from the kernel of the atomic set is R_{set}-ordered, where R_{set} holds for every two labels add(x) and add(y) with $x = y$.

A set of histories H is R-ordered iff every history in H is R-ordered. Let $\overline{H}$ be a library such that H is R-ordered. A match m of a history $h \in \overline{H}$ is *overwritten* by another match m' of h iff the labels of $+(m)$ and $+(m')$ are related by R, $+(m)$ finishes before $+(m')$, and every operation overlapping with $+(m')$ is completed.

Example 5.7. *Given the histories h_1 and h_2 in Figure 10, the match* $\{\text{write}(0), \text{read} \Rightarrow 0, \text{read} \Rightarrow 0\}$ *is overwritten by the match* $\{\text{write}(1), \text{read} \Rightarrow 1, \text{read} \Rightarrow 1\}$ *in h_2 but not in h_1 since a (*read $\Rightarrow 0$*) operation is pending in h_1.*

Given a match m overwritten by another match m' in $h \in \overline{H}$, every new completed operation o from an extension h' of h cannot be matched to $+(m)$, therefore m is also a match of h'. Otherwise, since all the operations overlapping with $+(m')$ are completed in h, o starts after $+(m')$ and H would contain a history where the operations $+(m)$, $+(m')$, o occur in this order. Therefore, H is not R-ordered.

Lemma 5.2. *Let H be a R-ordered, and $h_1, h_2 \in \overline{H}$. If $h_1 \rhd h_2$ and m is a match of h_1 overwritten by another match m' of h_1, then m is also a match of h_2.*

A match m of a history $h \in \overline{H}$ is called *obsolete* if (1) it is simply a match when the matching function of every history $h \in \overline{H}$ is injective or (2) it is overwritten by another match of h, when H is R-ordered. Lemmas 5.1 and 5.2, and Theorem 4 imply the following.

Corollary 3. *Let H be a match-removal closed history set such that M_h is injective for all $h \in \overline{H}$ or H is R-ordered, for some R. For every two histories $h_1, h_2 \in \overline{H}$ such that $h_1 \rhd h_2$ and m an obsolete match of h_1, $h_2 \setminus m$ is a history of $\overline{H}$.*

The following illustrates a possible source of incompleteness for monitoring algorithms which remove operations.

Example 5.8. *Figure 11 pictures two histories h_1 and h_2, the latter being an extension of the former. The (*rem $\Rightarrow 2$*)-operation is pending in h_1 and only completes in h_2. However, removing the match* $\{\text{add}(1), \text{rem} \Rightarrow 1\}$ *from h_1 before the pending* rem *completes results in a history admitted by the atomic stack. This highlights a risk of incompleteness which we must account for in our development of refinement-monitoring algorithms. Should we simply forget about the match* $\{\text{add}(1), \text{rem} \Rightarrow 1\}$ *while monitoring, we may not detect the violation obviated when the pending* rem *completes. In order to avoid such incompleteness, at the very least, our monitoring algorithm must remember that the* add(2) *operation should be ordered before* rem $\Rightarrow$ `empty` *(since* add(2) *must be ordered before* rem $\Rightarrow 1$*, which is ordered before* rem $\Rightarrow$ `empty`*), even after removing the match. Then when (*rem $\Rightarrow 2$*) completes, we could deduce that (*rem $\Rightarrow$ `empty`*) must be ordered after* add(2) *and before (*rem $\Rightarrow 2$*), a contradiction with the atomic stack theory* THEORY(H_{st}) *of Section 3.*

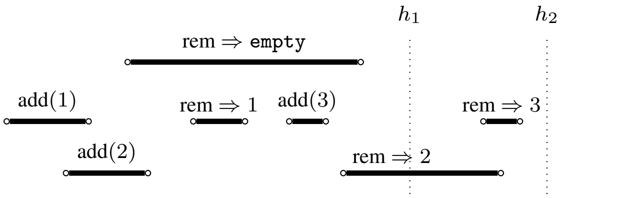


Figure 11. Two histories which are not admitted by the atomic stack, h_2 being an extension of h_1.

6. Empirical Validation

To demonstrate the practical value of the theory developed in the previous sections, we argue that our techniques

- scale far beyond existing algorithms, and
- are complete in practice.

To argue these points we have implemented three basic refinement-checking algorithms.

ENUMERATE is our implementation of the classical linearizability-checking algorithm [21] implemented by Line-up [6], checking each history h by enumerating the linearizations h' of h's completions. We check whether each h' is included in the kernel H by asking[1] whether $h' \models \text{THEORY}(H)$. As soon as this check succeeds, we conclude that $h \in \overline{H}$. Otherwise if this check fails for all linearizations, we conclude $h \notin \overline{H}$.

SYMBOLIC checks each history h by reduction to the satisfiability of $\text{STRONGER}(h) \wedge \text{THEORY}(H)$, as described in Section 4, delegating the enumeration of both completions and linearizations to an underlying solver. If the satisfiability check succeeds, or is inconclusive, we conclude that $h \in \overline{H}$. Otherwise if unsatisfiability is found, we conclude that $h \notin \overline{H}$.

SATURATE avoids the expensive propositional backtracking inherent to the aforementioned SYMBOLIC checker by limiting the satisfiability check to Boolean constraint propagation. Essentially, we implement a customized incremental solver which only saturates with unit propagation, avoiding any propositional branching. If a contradiction is found, we conclude that $h \notin \overline{H}$. Otherwise if saturation fails to reveal a contradiction, we conclude $h \in \overline{H}$.

Additionally, we have implemented the removal of obsolete matches as outlined in Section 5, which can be enabled for the SYMBOLIC and SATURATE algorithms.

We have studied ten concurrent data structure implementations from the Scal[2] High-Performance Multicore-Scalable Computing suite. Six of these implementations, such as the Michael-Scott Queue [16], are meant to preserve observational refinement[3] while the other four, such as the non-blocking bounded-reordering queue [13], are meant to preserve weaker properties.

The input to our checking algorithms are histories given as text files consisting of line-separated call and return actions. For the selected set of concurrent object implementations, we generated the histories of several executions under pseudo-random scheduling by logging calls and returns in the order in which they occurred.

[1] Classically this check is performed by set inclusion, as the length of h is assumed to be bounded by some number $n \in \mathbb{N}$ of operations, and the subset $H_n \subseteq H$ of n-operation histories of H is computable in finite time. As we assume no such bound on the number of operations, we perform this check via theorem-prover query instead.

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

[3] More precisely, they are designed to be linearizable.

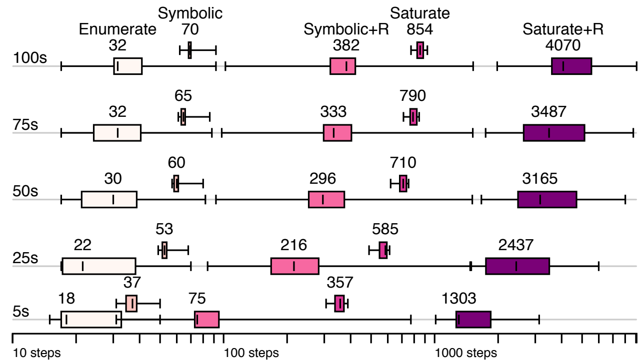


Figure 12. The number of steps each algorithm is able to process in given time limits of 5s, 25s, 50s, 75s, and 100s over 10 histories of Scal's Michael-Scott Queue implementation. Whiskers indicate minimums and maximums, numbers indicate medians, and box extents indicate first and third quartiles. Steps are plotted to a logarithmic scale.

While scanning an input history, the selected algorithm performs a membership test at each prefix at which an operation completes — i.e., at return actions.

Our first set of experiments (§6.1) demonstrates that our symbolic algorithms are drastically more scalable than existing algorithms, in that they are able to process vastly more history operations in much shorter time. Our second set of experiments (§6.2) demonstrates that our more efficient algorithms are complete in practice: the violations surfacing in the logs of actual executions are consistently discovered. We made all measurements on similar MacBook Pro 2.XGHz Intel Core i5/i7 machines, and discharged theorem-prover queries with an in-process instance of Z3[4].

Our implementation of the algorithms and all (generated) histories used in these experiments, are available on GitHub[5].

6.1 Scalability of Symbolic Checking

Our first experiment measures the number of steps each algorithm is able to process for varying time limits over 10 histories of Scal's Michael-Scott Queue implementation with 10000 steps each. We used five per-history time limits of 5s, 25s, 50s, 75s, and 100s. Results are shown in the graph of Figure 12 — results are similar for the other nine Scal implementations. The ENUMERATE algorithm performs worst, progressing only from median 18 steps in 5s to median 32 steps in 100s. The SYMBOLIC algorithm is a significant improvement, progressing from median 37 steps in 5s to median 70 steps in 100s. While adding match removal helps, achieving roughly an order-of-magnitude improvement over ENUMERATE, the cost of checking remains exponential in the number of steps.

Even without match removal, the SATURATE checker achieves a drastic improvement over ENUMERATE, progressing from median 357 steps in 5s to median 854 steps in 100s. Most impressively, adding match removal to the SATURATE checker allows it to process median 1303 steps in under 5s.

While the measurements of Figure 12 do demonstrate that SATURATE is more scalable than ENUMERATE and SYMBOLIC, they do not reveal whether SATURATE scales linearly in the number of steps when match removal is enabled. This is not visible since the asymptotic complexity of SATURATE grows polynomially in

[4] `https://github.com/Z3Prover/z3`

[5] `https://github.com/imdea-software/observational-refinement-checking`

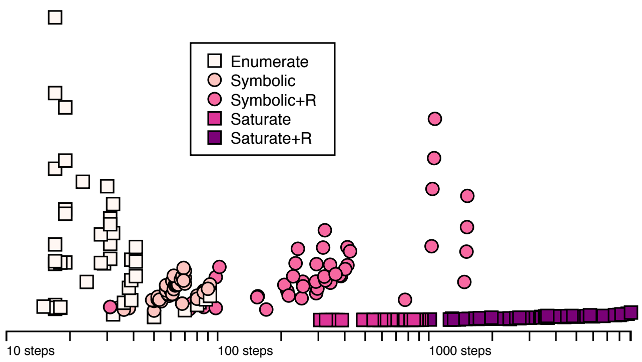


Figure 13. The number of steps each algorithm is able to process in time limits 5s, 25s, 50s, 75s, and 100s normalized by the square of average capacity of a given run over 10 histories of Scal's Michael-Scott Queue implementation. Steps are plotted to a logarithmic scale on the x-axis, and the y-axis represents normalized time.

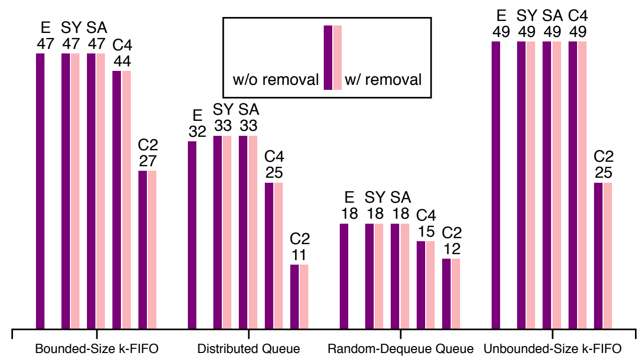


Figure 14. The number of violations each algorithm is able to detect across 100 histories of each of the four non-linearizable Scal implementations. Algorithms are abbreviated: ENUMERATE, SYMBOLIC, SATURATE, and COUNTING(k), for $k = 2, 4$.

the *capacity*[6] of the given concurrent data structures, and the capacities seen in our pseudo-random executions tend to grow as time goes on. Figure 13 cancels out the effect of capacity growth by normalizing data points against the square of the average capacity throughout a run: for each data point $\langle s, t, c\rangle$ of s steps in time t with average capacity c, we plot the point $\langle s, t/c^2\rangle$. Here we clearly see that SATURATE scales linearly in the number of steps when normalized against capacity-squared, whereas ENUMERATE and SYMBOLIC continue to scale poorly despite normalization. The apparent anomaly that SATURATE appears to scale linearly even with match removal disabled is explained by the fact that unmatched (add) operations, which could not have been removed in any case, tend to outnumber matched operations in these recorded histories.

6.2 Completeness in Practice

Our second experiment measures the amount of violations each algorithm is able discover across 100 histories of each of the ten Scal implementations used. Six of the ten are linearizable, and, correctly, no algorithm reported a violation therein. The amount of violations detected in the remaining four are plotted in Figure 14. The ENUMERATE algorithm is the baseline being the most-obviously complete algorithm, and detects all violations except 1 for which it exceeds a 10s timeout. As expected, the SYMBOLIC algorithm, also being theoretically complete, also detects all violations. Validating our hypotheses that the SATURATE algorithm and match removal are complete in practice, Figure 14 demonstrates that every single violation is also caught by SATURATE, and enabling removal furthermore does not cause missed violations.

In order to compare the precision of our algorithms with Bouajjani et al. [5]'s parameterized approximation algorithms, we also plot the number of violations caught using the $k = 4$ and $k = 2$ approximations. Essentially, their k-approximation abstract histories via weakening by forgetting ordering constraints such that the resulting order is a k-length interval order. As Figure 14 demonstrates, while small values of k can miss many violations, larger values of k can catch increasingly more, at the expense of additional runtime overhead. To avoid clutter in Figure 12, we did not plot their runtimes, though we remark that Bouajjani et al.'s algorithms perform on par with our SYMBOLIC algorithm.

Finally, we address the possible sources of incompleteness due to match removal discussed in Section 5. Recall the history h_1 and its extension h_2 of Figure 11 from Example 5.8. Since the SATURATE algorithm does not speculate on whether the pending rem operation might match the add of 2 or 3 (or both!), it will not detect the violation in h_1 until the pending rem operation completes. Simply removing the add-rem match of value 1 from h_1 before the pending rem operation completes would result in a non-violating history. However, by applying the stack-theory axioms THEORY(H_{st}) to completed operations before removing this match, the SATURATE algorithm infers the constraints

$$\mathrm{add}(2) < (\mathrm{rem} \Rightarrow 1) < (\mathrm{rem} \Rightarrow \mathtt{empty})$$

of which $\mathrm{add}(2) < (\mathrm{rem} \Rightarrow \mathtt{empty})$ persists after the match removal. Finally, when the pending rem-operation does complete, returning 2, the SATURATE algorithm derives a contradiction, since

$$\mathrm{add}(2) < (\mathrm{rem} \Rightarrow \mathtt{empty}) < (\mathrm{rem} \Rightarrow 2).$$

The incremental nature of the SATURATE algorithm thus avoids the practically-occurring sources of possible incompleteness due to match removal of which we are aware.

7. Discussion

In this work, we do not claim *theoretical completeness* of the SATURATE algorithm since we lack a formal completeness proof. Such a proof appears to be very challenging, and would seem to rely on yet-to-be-articulated assumptions on naturally-occurring concurrent objects. However, our empirical experience, reported in Section 6, suggests completeness, and we have no evidence to suggest that SATURATE is incomplete, even with operation removal.

While the ENUMERATE algorithm is complete by definition, it follows from Sections 3 and 4 that the SYMBOLIC algorithm is also complete without operation removal, and it follows from Section 5 that both ENUMERATE and SYMBOLIC are incomplete with operation removal.

Based on our experience in this and prior works [4, 5], we conjecture the theoretical completeness of SATURATE as well, with and without operation removal, for naturally-occurring objects like atomic stacks, queues, and locks, under the assumption that all operations eventually return. This would imply that the propositional backtracking thought to be inherent in linearizability is unnecessary, and that linearizability is polynomial-time checkable for typical concurrent objects. This insight has not been suggested by any previous works of which we are aware.

[6] Here *capacity* means the number of values stored inside the data structure at a given moment.

8. Related Work

Previous work on automated refinement verification focuses on *linearizability* [12], which is now known to be equivalent when considering atomic reference implementations [5, 9]. The theoretical limits of linearizability checking are well studied. While checking a single execution is NP-complete [10], checking all executions of a finite-state implementation is in EXPSPACE when the number of program threads is bounded [2], and undecidable otherwise [4].

Several semi-automated verification approaches rely on annotating method bodies with *linearization points* [1, 3, 8, 14, 17, 20, 23] to reduce the otherwise-exponential number of possible linearizations to one single linearization. These methods typically rely on programmer annotation, and do not admit conclusive evidence of a violation in the case of a failed proof. "Aspect-oriented proofs" reduce verification for certain atomic objects to a small set of simpler properties. Specifically, checking executions of atomic queue implementations against the theory $\text{THEORY}(H_q)$ of the atomic queue[7] *kernel* from Section 3 is sound and complete for complete histories [11].

Most automated approaches for detecting linearizability violations [6, 7, 21, 22] enumerate the exponentially-many linearizations of each execution, limiting them to executions with few operations, as observed in Section 6. Colt [18]'s approach mitigates this cost with programmer-annotated linearization points, as mentioned above, and ultimately suffers from the same problem: a failed proof only indicates incorrect annotation.

One closely-related work aims to reduce the complexity of refinement checking by approximating executions with weaker, bounded histories [5]. While this approach is also sound, in the sense that only actual violations are flagged, its completeness in practice is reported to rely on observing *many* executions with varying thread schedules, and its applicability is limited to executions with a bounded number of argument/return values. On the contrary, the requirements for long-term execution monitoring which we address demand completeness for each individual execution without bounding the number of data values. Technically, our approximations differ as well: while theirs forgets the order between some operations, our match removal optimization forgets operations completely. Note however that by removing operations only *after* saturation of their logical implications, the effect of forgotten operations can persist.

9. Acknowledgements

This work is supported in part by the VECOLIB project (ANR-14-CE28-0018), the N-GREENS Software project (Ref. S2013/ICE-2731), and AMAROUT-II (EU-FP7-COFUND-291803).

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 *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, volume 7795 of *LNCS*, pages 324–338. Springer, 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 *Proc. 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

[4] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *Proc. 22nd European Symposium on Programming (ESOP '13)*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable refinement checking for concurrent objects. In *Proc. 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*, pages 651–662. ACM, 2015.

[6] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 330–340. ACM, 2010.

[7] J. Burnim, G. C. Necula, and K. Sen. Specifying and checking semantic atomicity for multithreaded programs. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 79–90. ACM, 2011.

[8] C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *Proc. 25th International Conference on Computer Aided Verification (CAV '13)*, volume 8044 of *LNCS*, pages 174–190. Springer, 2013.

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

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

[11] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *Proc. 24th International Conference on Concurrency Theory (CONCUR '13)*, volume 8052 of *LNCS*, pages 242–256. Springer, 2013.

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

[13] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *Proc. 12th International Conference on Parallel Computing Technologies (PaCT 13)*, volume 7979 of *LNCS*, pages 208–223. Springer, 2013.

[14] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In *Proc. Second World Congress on Formal Methods (FM '09)*, volume 5850 of *LNCS*, pages 321–337. Springer, 2009.

[15] M. M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM Thomas J. Watson Research Center, January 2004.

[16] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275. ACM, 1996.

[17] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *Proc. 29th Annual ACM Symposium on Principles of Distributed Computing (PODC '10)*, pages 85–94. ACM, 2010.

[18] O. Shacham, N. G. Bronson, A. Aiken, M. Sagiv, M. T. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Proc. 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 51–64. ACM, 2011.

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

[20] V. Vafeiadis. Automatically proving linearizability. In *Proc. 22nd International Conference on Computer Aided Verification (CAV '10)*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.

[21] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993.

[22] L. Zhang, A. Chattopadhyay, and C. Wang. Round-up: Runtime checking quasi linearizability of concurrent data structures. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*, pages 4–14. IEEE, 2013.

[23] S. J. Zhang. Scalable automatic linearizability checking. In *Proc. 33rd International Conference on Software Engineering (ICSE '11)*, pages 1185–1187. ACM, 2011.

[7] Technically, the axioms of this theory without our totality axiom.