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 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-invoke 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.
We begin by formalizing observational refinement (§). Violations are caught despite the possibility for incompleteness. Empirical validation that our violation-detection optimizations are profoundly-more scalable than existing techniques. We also continue to catch practically-occurring violations. Example 2.1. The history of the above execution. (c) A weaker history.

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 (§).
- A reduction from refinement-violation detection for single executions to propositional logic satisfiability (§).
- A sound theory of matched-operation removal, allowing the scalability required for runtime monitoring (§).
- Empirical validation that our violation-detection optimizations are scalable and effectively-complete (§).

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

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.

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 $z = \text{EMPTY}$ 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-befores of library method invocations. Fixing arbitrary sets $\Omega$, $M$, and $V$ of operation identifiers, method names, and parameter/return values, respectively, we define the set of operation labels as $L = (M \times \forall \times (V \cup \{\bot\}))$. We write $m(y) \Rightarrow v$ to denote the label $\ell = (m, u, v)$. When $v = \bot$ we say $\ell$ is pending, and otherwise completed. A history $h = \langle O, <, f \rangle$ is a partial order $<$ on a set $O \subseteq \Omega$ of operations labeled by $f : O \rightarrow L$ such that operations with pending labels are maximal. An operation $o$ with label $\ell$ is an $t$-operation, and $o$ is pending/completed when $\ell$ 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 ⇒ 3, since the push(1)-operation returns before the (pop ⇒ 3)-operation is called, the (pop ⇒ 1)-operation is incomparable to pop ⇒ 3, since it returns after the (pop ⇒ 3)-operation is called.

```
struct node *Top;

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

struct node *Top;

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

void Thread1();
    int data;
    struct node *next;

void Thread2();
    int x = 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.

Figure 2. An execution and its history.
This notion of histories gives rise to a natural weaker-than relation \( \sqsubseteq \) 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, \((O_1, <_1, f_1) \sqsubseteq (O_2, <_2, f_2)\) iff there exists an injection \( g : O_2 \to O_1 \) such that

- \( o \in \text{ran}(g) \) when \( f_1(o) = m(u) \Rightarrow v \) and \( v \neq \perp \),
- \( g(o_1) < g(o_2) \) implies \( o_1 < o_2 \) for each \( o_1, o_2 \in O_2 \),
- \( f_1(g(o)) \preceq f_2(o) \) for each \( o \in O_2 \).

where \((m_1(u_1) \Rightarrow v_1) \preceq (m_2(u_2) \Rightarrow v_2)\) iff \( m_1 = m_2, v_1 \equiv u_2, \) and \( v_1 \in \{v_2, \perp\} \). When the injection \( g \) need be fixed, we write \( h_1 \preceq h_2 \). 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 4-5 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 \text{meth}\(-\)-od, \text{arg}\(-\)-ment, and \text{ret-\}-\text{urn}, as well as the predicate symbols match.
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 (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

$$\text{ATOMIC} \land \text{INCLUDE} \land \text{EXCLUDE}$$

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

$$\text{ATOMIC} \land \text{Total}(\text{lock}, \text{unlock}) \land \text{INJECTIVE}$$

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

$$\text{ATOMIC} \land \text{Total}(\text{acquire}, \text{release}) \land \text{LIMIT}$$

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 $v' \neq \text{null}$ if it has paired up with an exchange($v'$)

**Figure 4.** Formulas for collection objects.

**INCLUDE** and **EXCLUDE** specify when a contains-operation may return true. We represent the kernel $H_r$ of atomic sets by

$$\text{ATOMIC} \land \text{INCLUDE} \land \text{EXCLUDE}$$

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 (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

$$\text{ATOMIC} \land \text{READWRITE} \land \text{READFROM}$$

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

$$\text{ATOMIC} \land \text{TOTAL}(\text{lock}, \text{unlock}) \land \text{INJECTIVE}$$

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

$$\text{ATOMIC} \land \text{TOTAL}(\text{acquire}, \text{release}) \land \text{LIMIT}$$

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 $v' \neq \text{null}$ if it has paired up with an exchange($v'$)

**Figure 5.** Formulas for set objects.

**READWRITE**

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

**READFROM**

$$\forall w_1. \text{meth}(w_1) = \text{write} \land \text{meth}(r) = \text{read} \land \lnot \text{match}(w_1, r) \land b(w_1, r) \land b(w_2, r)$$

**Figure 6.** Formulas for register objects.

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

**COMPLETED** $\text{A EXCHANGE}$

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 $h \in \mathcal{T}$ reduces to propositional satisfiability, given a formula $\text{THEORY}(H)$ characterizing the histories of the library kernel $H$. Note that $h \in \mathcal{T}$ iff $h$ is weaker than some history $h' \in H$, or equivalently weaker than some history $h'$ such that $h', M \vdash \text{THEORY}(H)$, 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 $\text{STRONGER}(h)$ characterizing the histories stronger than $h$. Together
with \( \text{Theory}(H) \), this formula describes all stronger histories satisfying \( \text{Theory}(H) \), and is therefore equivalent to \( h \in \mathcal{T} \). 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 \( \text{Theory}(H) \) and \( \text{Stronger}(h) \) 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 \( g : O' \rightarrow O \) such that:

1. operations related by \( g \) have the same label, i.e., for each \( o \in O \), \( f(o) = f'(g^{-1}(o)) \), and
2. 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 \( \text{Stronger}(h) \), defined as the conjunction

\[
\text{Domain}(h) \land \text{Labels}(h) \land \text{Order}(h)
\]

using the formulas of Figure 8 characterizing the identifiers, labels, and order constraints of \( h \). Note that the \( \text{Domain}(h) \) formula restricts the interpretation domain of each variable to the operations of \( h \). As a consequence of Lemma 4.1, the formula \( \text{Stronger}(h) \) 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 \( h' \models \text{Stronger}(h) \).

It follows that the library membership test \( h \in \mathcal{T} \) for complete histories \( h \) reduces to first-order satisfiability.

**Theorem 1.** Let \( h \) be a complete history, and \( H \) a history set. Then \( h \in \mathcal{T} \) iff \( \text{Stronger}(h) \land \text{Theory}(H) \) is satisfiable.

As long as the formula \( \text{Theory}(H) \) contains only a fixed set of predicates e.g., = and \( \leq \), as is the case for all formulas of Figures 3–7, satisfiability of \( \text{Stronger}(h) \land \text{Theory}(H) \) 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 \( \text{[Stronger}(h) \land \text{Theory}(H)]} \) by replacing each universally-quantified subformula \( \forall x, \phi \) by \( \bigwedge_{o \in O} \phi[x \mapsto o] \), and each existentially-quantified subformula \( \exists x, \phi \) by \( \bigvee_{o \in O} \phi[x \mapsto o] \).

It follows that this propositional formula is equisatisfiable to the original first-order formula, and is constructed in polynomial time.

**Corollary 1.** Let \( h \) be a complete history, and \( H \) a history set. Then \( h \in \mathcal{T} \) iff the propositional formula

\[
[\text{Stronger}(h) \land \text{Theory}(H)]
\]

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 \( \text{Stronger}(h) \) 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 \( \perp \) 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 \( \text{Stronger}(h) \) formula for incomplete histories \( h \) as the conjunction

\[
\text{Domain}(h) \land \text{Labels}(h) \land \text{Order}(h) \land \text{Used},
\]

where Used is defined in Figure 8.

For arbitrary, not necessarily complete, histories \( h \), the models of \( \text{Stronger}(h) \) are histories paired with an interpretation \( U : O \rightarrow \mathcal{B} \) for the domain predicate used, mapping the operations of \( O \) to \{true, false\}. Given such a model \( (h, U) \), 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 \( U : O' \rightarrow \mathcal{B} \). Then \( h', U \models \text{Stronger}(h) \) iff \( h < U(h') \).

We leverage the predicate used to guard the domain of quantifiers in \( \text{Theory}(H) \). For simplicity, we assume that \( \text{Theory}(H) \) 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 \( \varphi = \forall x. \exists y. \psi \) as

\[
G(\varphi) = \forall x. \exists y. \bigwedge_{i} \text{used}(x_i) \Rightarrow \bigwedge_{j} \text{used}(y_j) \land \psi
\]

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 \( h \in \mathcal{T} \) iff the first-order formula

\[
\text{Stronger}(h) \land G(\text{Theory}(H))
\]

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 \( h \in \mathcal{T} \) iff the propositional formula

\[
[\text{Stronger}(h) \land G(\text{Theory}(H))]
\]

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 9 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 \( h \in \mathcal{T} \) iff there exists a history \( h' \) stronger than \( h \) and a matching function \( M : O \rightarrow O \) such that \( h', M = \text{Theory}(H) \). We assume in the following that the matching function \( M : O \rightarrow O \) of every history \( h \) is uniquely determined by the operation labels in \( h \), i.e., there exists a function \( M : L \rightarrow L \) such that \( M(o) = o' \) iff \( M(f(o)) = f(o') \). 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 \( h = (O, <, f) \) 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 5 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 \( \mathcal{T} \). Therefore, if by deleting matches from a history we get a history which is not admitted by a library \( \mathcal{T} \) then the initial history is also not admitted by \( \mathcal{T} \).

Theorem 4. \( \mathcal{T} \) is match-removal closed if \( H \) is.

\[
\text{Proof.} \quad \text{Let } h \in \mathcal{T} \text{ and } m \text{ a match of } h. \text{ By definition, there exists a history } h' \in H \text{ such that } h \preceq h'. \text{ Since the matching functions are uniquely determined by the operation labels, } m \text{ is also a match of } h'. \text{ By hypothesis, } H \text{ is match-removal closed which implies that the history } h'' \text{ obtained from } h' \text{ by deleting the operations in } m \text{ is also in } H. \text{ From the definition of } \preceq \text{ it follows that the history } h \setminus m \text{ is weaker than } h'' \text{ which implies } h \setminus m \in \mathcal{T}. \]

While the statement of Theorem 4 implies that a history \( h \setminus m \), where \( m \) is a match of \( h \), belongs to \( \mathcal{T} \) whenever \( h \in \mathcal{T} \), 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.
write(0) | write(1) | read ⇒ 1 | read ⇒ 1 | h₁ | h₂
read ⇒ 0 | read ⇒ 0

Figure 10. Two histories h₁ and h₂ of the atomic register, where h₂ is an extension of h₁.

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 is ordered in both h₁ and h₂. Therefore, 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 h₂ extends h₁ iff h₂ is the history of an execution that extends the execution represented by h₁. Formally, h₂ = (O₂, <₂, f₂) extends h₁ = (O₁, <₁, f₁), written h₁ ⊢ h₂, iff

- O₁ ⊆ O₂,
- f₁(a) ≤ f₂(a) for each a ∈ O₁,
- a₁ <₁ a₂ iff a₁ <₂ a₂ for each a₁, a₂ ∈ O₁, and
- a₁ <₂ a₂ for each a₁ ∈ O₁ and a₂ ∈ O₂ \ O₁.

Lemma 5.1. Let ℋ be a library such that M₀ is injective for all h ∈ ℋ. For every two histories h₁, h₂ ∈ ℋ such that h₁ ⊢ h₂, if m is a match of h₁ then m is also a match of h₂.

Example 5.6. Figure 10 pictures two histories h₁ and h₂ of the atomic register, the latter being an extension of the former. We assume that the matching function maps every (read ⇒ i) operation to the write(i) operation. The match {write(1), read ⇒ 1} of h₁ is strictly included in the match {write(1), read ⇒ 1, read ⇒ 1} of h₂, and therefore not a match of h₂. Removing the match of h₂ and extending it with the (read ⇒ 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 m₁ and m₂ of h such that R(+(m₁), +(m₂)), each operation of m₁ is ordered before each operation of m₂.

Consider the atomic register and the relation Read 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 register is R-ordered, where Read 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 ℋ be a library such that H is R-ordered. A match m of a history h ∈ 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 h₁ and h₂ in Figure 7, the match {write(0), read ⇒ 0, read ⇒ 0} is overwritten by the match {write(1), read ⇒ 1, read ⇒ 1} in h₂ but not in h₁ since a (read ⇒ 0) operation is pending in h₁.

Given a match m overwritten by another match m’ in h ∈ ℋ, every 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 h₁, h₂ ∈ ℋ. If h₁ ⊢ h₂ and m is a match of h₂ overwritten by another match m’ of h₁, then m is also a match of h₂.

A match m of a history h ∈ ℋ is called obsolete if (1) it is simply a match when the matching function of every history h ∈ 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 M₀ is injective for all h ∈ ℋ or H is R-ordered, for some R. For every two histories h₁, h₂ ∈ ℋ such that h₁ ⊢ h₂ and m is an obsolete match of h₁, h₂ \ m is a history of ℋ.

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

Example 5.8. Figure 7 pictures two histories h₁ and h₂, the latter being an extension of the former. The (rem ⇒ 2)-operation is pending in h₁ and only completes in h₂. However, removing the match {add(1), rem ⇒ 1} from h₁ 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 {add(1), rem ⇒ 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 ⇒ empty (since add(2) must be ordered before rem ⇒ 1, which is ordered before rem ⇒ empty), even after removing the match. Then when (rem ⇒ 2) completes, we could deduce that (rem ⇒ empty) must be ordered after add(2) and before (rem ⇒ 2), a contradiction with the atomic stack theory Theory(H₂) of Section 3.
Classically this check is performed by set inclusion, as the length of
To argue these points we have implemented three basic refinement-
More precisely, they are designed to be linearizable.
assumed no such bound on the number of operations, we perform this check
selected set of concurrent object implementations, we generated
Scott Queue [16], are meant to preserve observational refinement
The input to our checking algorithms are histories given as text
the asymptotic complexity of SATURATE grows polynomially in
While scanning an input history, the selected algorithm performs a
Our implementation of the algorithms and all (generated) histories
6.1 Scalability of Symbolic Checking
Our first experiment measures the number of steps each algorithm
We made all measurements on similar MacBook Pro 2.4GHz Intel
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 2.4GHz Intel
Core i5/7 machines, and discharged theorem-prover queries with
an in-process instance of Z3
Our implementation of the algorithms and all (generated) histories
used in these experiments, are available on GitHub

---

1Classically this check is performed by set inclusion, as the length of h is assumed to be bounded by some number n ∈ N of operations, and the subset Hn ⊆ 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.
2http://scal.cs.uni-salzburg.at
3More precisely, they are designed to be linearizable.

---

### Figure 11.
Two histories which are not admitted by the atomic stack, h2 being an extension of h1.

### 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.

---

# 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 whether h’ ⊆ THEORY(H). As soon as this check succeeds, we conclude that h ∈ H. Otherwise if this check fails for all linearizations, we conclude h ∉ H.

**SYMBOLIC** checks each history h by reduction to the satisfiability of STRONGER(h)∧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 ∈ H. Otherwise if unsatisfiability is found, we conclude that h ∉ 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 ∉ H. Otherwise if saturation fails to reveal a contradiction, we conclude h ∈ 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 [15], are meant to preserve observational refinement[6] 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.
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 $(s, t, c)$ of $s$ steps in time $t$ with average capacity $c$, we plot the point $(s, t/c^2)$. 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 to 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 14 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 5.2. Recall the history $h_1$ and its extension $h_2$ of Figure 11 from Example 5.3. 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 THEOREM $(H_a)$ to completed operations before removing this match, the SATURATE algorithm infers the constraints

$$\text{add}(2) \prec \text{rem} \Rightarrow 1 \prec \text{rem} \Rightarrow \text{empty}$$

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

$$\text{add}(2) \prec \text{rem} \Rightarrow \text{empty} \prec \text{rem} \Rightarrow 2.$$ 

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 5.3 and 5.4 that the SYMBOLIC algorithm is also complete without operation removal, and it follows from Section 5.5 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 [3].

Several semi-automated verification approaches rely on annotating method bodies with linearization points [1][9][14][17][22] 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 THEORY(Hq) of the atomic queue kernel from Section 3 is sound and complete for complete histories [11].

Most automated approaches for detecting linearizability violations [6][7][21][23] 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

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


