Weak-Consistency Specification via Visibility Relaxation

MICHAEL EMMI, SRI International, USA
CONSTANTIN ENEA, IRIF, Univ. Paris Diderot & CNRS, France

Effective software specifications enable modular reasoning, allowing clients to establish program properties without knowing the details of module implementations. While some modules' operations behave atomically, others admit weaker consistencies to increase performance. Consequently, since current methodologies do not capture the guarantees provided by operations of varying non-atomic consistencies, specifications are ineffective, forfeiting the ability to establish properties of programs that invoke non-atomic operations.

In this work we develop a methodology for specifying software modules whose operations satisfy multiple distinct consistency levels. In particular, we develop a simple annotation language for specifying weakly-consistent operations via *visibility relaxation*, wherein annotations impose varying constraints on the visibility among operations. To integrate with modern software platforms, we identify a novel characterization of consistency called *sequential happens-before consistency*, which admits effective validation. Empirically, we demonstrate the efficacy of our approach by deriving and validating relaxed-visibility specifications for Java concurrent objects. Furthermore, we demonstrate an optimality of our annotation language, empirically, by establishing that even finer-grained languages do not capture stronger specifications for Java objects.

CCS Concepts: • **Theory of computation → Program specifications**; • **Computing methodologies →** *Concurrent computing methodologies*; • **Software and its engineering →** *Specification languages*;

Additional Key Words and Phrases: concurrency, specification, weak-consistency

ACM Reference Format:
Michael Emmi and Constantin Enea. 2019. Weak-Consistency Specification via Visibility Relaxation. *Proc. ACM Program. Lang.* 3, POPL, Article 60 (January 2019), 28 pages. https://doi.org/10.1145/3290373

1 INTRODUCTION

Many software platforms enable high-performance multithreaded code by providing optimized *concurrent objects* which encapsulate lock-free shared memory access protocols into high-level abstract data types. For instance, The JDK provides 16 atomic primitive register types, e.g., with atomic increment methods, and 14 concurrent data structures, e.g., with atomic offer, peek, and poll methods. Having been designed and implemented by experts, and scrutinized by a large community of JDK users, these concurrent objects offer high performance and reliability.

Given the potentially-enormous amount of software that relies on these concurrent objects, it is important to maintain precise specifications and ensure that implementations adhere to their specifications. Many methods are expected to behave atomically, meaning that the results of concurrently-executed invocations match the results of some serial execution of those same invocations; they are expected to behave atomically despite the heavy optimizations employed to avoid blocking and exploit parallelism, e.g., by preferential use of specialized machine instructions like atomic compare and exchange instead of lock-based synchronization.

Authors' addresses: Michael Emmi, SRI International, USA, michael.emmi@sri.com; Constantin Enea, IRIF, Univ. Paris Diderot & CNRS, France, cenea@irif.fr.

2475-1421/2019/1-ART60
https://doi.org/10.1145/3290373

While lock-free implementations can increase performance substantially, the insistence on atomicity generally imposes fundamental synchronization bottlenecks [Gilbert and Lynch 2002]. Therefore, some methods, like the iterator methods of JDK concurrent data structures, embrace consistency criteria weaker than atomicity. While relaxation from atomicity to "weak consistency" provides wiggle room for performance optimization, it impedes modular reasoning. Unlike the de facto guarantee of atomicity which is given a precise formal meaning via *linearizability* [Herlihy and Wing 1990], the guarantees provided by "weakly consistent" operations are unclear since there are many ways in which consistency can be weakened.

For instance, an invocation of a set's weakly-consistent size method in two parallel threads { add(1); remove(2) } and { add(2); size() $\Rightarrow n$ } may return $n = 0$ in certain executions where invocations interleave.[1] This outcome is not admissible by atomic operations, since $n > 0$ in every linearization: $n = 2$ if size executes between the first thread's operations; otherwise $n = 1$. Our intuition of "weakly-consistent" may admit $n = 0$ in the example above, by considering executions where size interleaves with both the first thread's operations, yet only observes the effect of remove. However, our intuition would probably not admit $n = 100$ nor $n = -1$. Yet, without identifying a precise semantics to "weak consistency" we forfeit all guarantees, making client reasoning impossible.

In this work we investigate a methodology for the precise specification of concurrent objects with weakly-consistent operations. Our starting point is existing methodologies for axiomatic consistency specification of concurrent objects which consider operation visibility [Burckhardt et al. 2014; Perrin et al. 2016]. This approach essentially extends the linearizability-based specification methodology with a dynamic *visibility* relation among operations, in addition to the standard dynamic *real-time order* and *linearization* relation. Permitting weaker visibility relations models outcomes in which an operation may not observe the effects of operations that are linearized before it. For instance, while $n = 1$ in the full linearization add(1); add(2); remove(2); size() $\Rightarrow n$ of the example above, relaxing the visibility of size admits the $n = 0$ outcome, in the partially-visible linearization add(2); remove(2); size() $\Rightarrow n$ where the first add operation is not visible to size.

While this axiomatic framework enables the precise characterization of weak consistencies, its general application to software API annotation remains an open problem for two reasons. First, precise annotation requires a fine-grained categorization of consistency relaxations so that precise consistencies can be declared and relied upon; previous works elaborate only a few categories of eventual and causal consistency [Burckhardt et al. 2014; Perrin et al. 2016]. Neither eventual nor causal consistency precisely capture the $n = 0$ outcome in the example above; causal consistency doesn't allow size to observe the first thread's remove operation without also observing its add operation; eventual consistency does not constrain n at all in this case. Second, annotating API methods requires mixtures of consistency levels since APIs generally include methods of varying consistencies; previous works, e.g., [Batty et al. 2013; Burckhardt et al. 2014; Doherty et al. 2018; Dongol et al. 2018; Herlihy and Wing 1990; Perrin et al. 2016], develop only global consistency levels that apply to all API methods. For instance, a set's add and remove methods may guarantee atomicity even if the size method does not. The mixture poses a challenge to specification since the consistencies of individual methods like size generally depend on the other methods' implementations.

Accordingly, we develop an annotation-based fine-grained consistency specification methodology for software APIs. In particular, we identify *visibility relaxation* as a key mechanism for consistency weakening, and several levels of per-method visibility, depicted in Figure 1. In order of decreasing relaxation, where each level includes the guarantees of previous levels:

(1) *weak visibility* does not constrain the operations visible to a given operation;

[1]This outcome is observable in Java's ConcurrentSkipListSet.

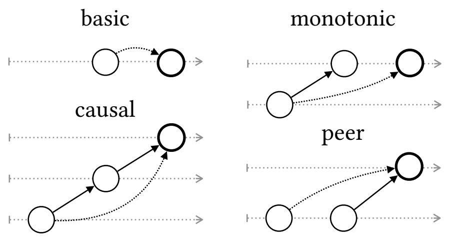

Fig. 1. The operations (circles) visible to the given operation (bold circle) depend on its annotation. The *basic* annotation implies visibility (dashed arrow) of happens-before predecessors (horizontal line), while *monotonic* implies visibility of those visible (solid arrow) to predecessors. The *peer* annotation implies visibility of predecessors of visible operations, and *causal* implies transitive visibility.

ConcurrentHashMap: contains
`{ put(1,0); contains(0) } || { put(0,0); put(1,1) }`

outcome	atomic?	frequency
`1, true, null, null`	✓	770,621
`null, true, null, 0`	✓	2,074,326
`null, **false**, null, 0`	×	3
`1, **false**, null, null`	×	0

Fig. 2. The contains method of Java's ConcurrentHashMap implementation exhibits non-atomic outcomes. These outcomes are observed with the given frequencies during 1 second of stress testing the given program, in which two threads invoke the put and contains methods. Each outcome is given as a list of return values following the order of the invocations in the program, e.g., the first value in each outcome is the return value of `put(1,0)`.

(2) *basic visibility* requires each operation to see its happens-before predecessors;[2]
(3) *monotonic visibility* requires each operation to see those its happens-before predecessors see;
(4) *peer visibility* requires each operation to see the happens-before predecessors of those it sees;
(5) *causal visibility* requires each operation to see those visible to those it sees;
(6) *complete visibility* requires each operation to see those linearized before it.

By distilling each level into a per-method annotation, we devise an annotation language capable of specifying varying yet precise weak-consistency guarantees among API operations, e.g., allowing only certain operations to exercise incomplete visibility. For instance, the $n = 0$ outcome of the size method above is inconsistent even with peer visibility, since size must not see the predecessor, add(1), of an operation it sees, remove(2). However, this outcome is consistent with monotonic visibility, even when add and remove have complete visibility, so long as add(2) is concurrent with add(1), and can thus be linearized before. Unlike the informal annotation of "weakly consistent" which guarantees nothing, ascribing monotonic visibility to the size method guarantees $0 \leq n \leq 2$.

Our annotation language enables fine-grained mixed-consistency specification and validation for popular software APIs, a claim we substantiate by deriving specifications for Java concurrent objects using a novel algorithm which combines test generation and stress testing. We find that the majority of Java methods have complete or monotonic visibility, suggesting that Java's informal notion of "weakly consistent" corresponds to monotonic visibility. Furthermore, this suggests that there are several additional opportunities for potential optimization, by adopting relaxed specifications which embrace the spectrum of consistency levels our annotations provide.

Of particular technical interest, we introduce novel characterization of linearization-based consistency specifications we call *(relaxed) sequential happens-before consistency*, or (R)SHBC. Unlike the classical linearizability criterion, which admits only linearizations consistent with *real-time invocation order*,[3] SHBC admits any linearization consistent with the weaker platform-defined happens-before invocation order, thus including a superset of those admitted classically. For instance, if an atomic size operation of the thread { add(2); size() $\Rightarrow n$ } executes between the

[2]The meaning of *happens-before* is platform dependent, yet generally includes *program order*, i.e., the static order among a thread's invocations, as well as *synchronizes-with order*, i.e., the dynamic order among conflicting synchronization primitives.
[3]The real-time order includes all pairs of operations where the first's return action executes prior to the second's call action.

operations of the thread { add(1); remove(2) }, then in both possible linearizations, resolving the order of add(1) and add(2), its outcome must be $n = 2$; since SHBC admits additional linearizations, in which size need not be linearized between the second thread's operations, both $n = 1$ and $n = 2$ outcomes are admitted. While ignoring the real-time order is unsound for validating linearizability, we argue that sound runtime verification of linearizability is anyhow impossible in practice, for the same reason, given the lack of effective means to record real-time orders precisely without interfering with, e.g., weak-memory behaviors of the monitored implementation. Despite this lack of per-execution precision, soundness is recovered in the limit over all executions, as long as any given real-time constraint can be enforced by some program, e.g., using synchronization. Of more practical concern, we demonstrate that (R)SHBC is better adapted to the specifications of modern software platforms, which generally eschew the mention of real-time ordering guarantees to allow implementation flexibility. Furthermore, validation can be performed with great efficiency, e.g., compared to linearizability checking, using off-the-shelf testing tools. This efficiency allows us to effectively discover violations to incorrect consistency annotations in Java concurrent objects.

To summarize, the contributions of this work are fourfold:

- we develop an annotation language for describing the guarantees of software APIs with operations of varying relaxed-visibility consistency (§3);
- we develop a novel (relaxed) sequential happens-before consistency to adapt linearization-based atomicity criteria to modern platforms like Java and C++ (§4);
- we conduct an empirical study deriving and validating relaxed-visibility consistency annotations in Java concurrent objects (§5);
- we demonstrate that our annotation language is optimal, in a sense, by showing that the derived specifications are no weaker than specifications derived from a much more expressive language of consistency relaxation (§6).

Combined, these contributions form a simple and effective specification methodology for weakly-consistent operations which is applicable to modern platforms like Java and C++. Furthermore, they outline the foundational principles for developing weak-consistency specification mechanisms for other platforms, to which alternate consistency models may apply. To the best of our knowledge, we are the first to develop a generic methodology capable of specifying arbitrary software APIs with operations of varying consistencies, despite their prevalence in practice, e.g., in Java.

Aside from the sections mentioned above, we begin by motivating our development in Section 2, and end by discussing related work and conclusions in Sections 7 and 8.

2 MOTIVATION

Precise consistency guarantees are not typically attached to operations in today's software APIs. Instead, software API specifications suggest vague guarantees, typically distinguishing whether operations are "atomic" or "locking." For instance the Java Platform Standard Edition 10 API Specification states:[4]

> *A concurrent collection is thread-safe, but not governed by a single exclusion lock.*

and describes a ConcurrentMap as

> *A Map providing thread safety and atomicity guarantees.*

While these statements suggest that the operations of ConcurrentMaps are atomic and lock-free, this is later undermined, e.g., in the API specification for ConcurrentSkipListMap which implements ConcurrentMap:[5]

[4]Package java.util.concurrent: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/package-summary.html
[5]Class ConcurrentSkipListMap: https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ConcurrentSkipListMap.html

ConcurrentHashMap: size

`{ put(1,0); put(1,1); size() } || { remove(1) }`

outcome	atomic?	frequency
`null, 0, 0, 1`	✓	949
`null, 0, 1, 1`	✓	746,263
`null, 0, 1, null`	✓	2,614,780
`null, null, 1, 0`	✓	14,833
`null, null, **2**, 0`	×	35

Fig. 3. A non-atomic outcome for the size method. Each outcome is given as a list of return values following the order of the invocations in the program, e.g., the first value in each outcome is the return value of `put(1,0)`.

ConcurrentHashMap: isEmpty

`{ put(1,1) } || { put(1,2); isEmpty() }`

outcome	atomic?	frequency
`2, null, false`	✓	680,432
`null, 1, false`	✓	3,456,201
`null, 1, **true**`	×	57

Fig. 4. A non-atomic outcome for isEmpty. Each outcome is given as a list of return values following the order of the invocations in the program, e.g., the first value in each outcome is the return value of `put(1,1)`.

> *Iterators and spliterators are* weakly consistent.

whose meaning is elaborated elsewhere:

> *They are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.*

The specification proceeds to undermine the atomicity guarantees of several other operations:[6]

> *Beware that, unlike in most collections, the size method is not a constant-time operation. Because of the asynchronous nature of these maps, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal. Additionally, the bulk operations putAll, equals, toArray, containsValue, and clear are not guaranteed to be performed atomically. For example, an iterator operating concurrently with a putAll operation might view only some of the added elements.*

The specifications above are ambiguous about the consistency guarantees provided by various operations. While one expects the core Map operations like put, get, remove, and containsKey to be atomic and lock-free, it is unclear what to expect from the others since the specification undermines guarantees for many operations, and "weakly consistent" provides only broad guarantees. Besides those broad guarantees on iterators, non-atomic operations would provide absolutely no guarantees, effectively sabotaging any client-side reasoning, and enabling even nonsensical optimizations.

Many other operations are not atomic. Figures 2–4 list a few multi-threaded programs invoking methods of Java's ConcurrentHashMap, and their corresponding outcomes as observed during one second of stress testing each. ConcurrentHashMap's get and put methods are believed to behave atomically, and so we focus on the apparently non-atomic methods contains, size, and isEmpty.

An atomic implementation of contains, as invoked in Figure 2, should never return false, since the value 0 is present during the invocation of contains in every linearization: if the entry $\langle 1, 1\rangle$ has overwritten $\langle 1, 0\rangle$, then the entry $\langle 0, 0\rangle$ must also be present. However, Java's implementation allows the non-atomic outcome where contains returns false, in order to avoid the expensive synchronization otherwise imposed by examining all map entries atomically.[7] Intuitively, contains can begin executing before key 0 is present, stepping over its hash bucket, and proceed to the hash

[6]The first two sentences are present only up to Version 9 of the Java Platform Specification.

[7]All empirical observations are drawn from the Java Platform Standard Edition 9.

bucket for key 1. If interrupted by the second thread, the entry $\langle 1, 0\rangle$ can be replaced by $\langle 1, 1\rangle$ before contains resumes, in which case contains does not find value 0, and returns false.

To enable precise reasoning about the admittance of behaviors which are non-atomic by design, we propose a principled mechanism called *visibility relaxation* to weakening atomicity. This mechanism essentially allows operations to behave as if they do not observe the effects of certain other operations. To impose guarantees in the presence of relaxed visibilities, we introduce per-method annotations which constrain the operations whose effects an invocation of the given method must observe. For instance, the *complete* visibility forces the observation of all linearized-before operations. When all methods have complete visibility, only atomic behaviors are admitted. Figure 1 illustrates the semantics of visibility annotations graphically, and Section 3 describes them precisely.

To admit the aforementioned non-atomic outcome of the contains method in Figure 2, we may relax contains' visibility to account for the fact that it might not observe concurrently-inserted entries, e.g., $\langle 0, 0\rangle$. The *monotonic* visibility, which requires that contains see its happens-before predecessors, i.e., put(1,0), along with any operations seen by them, admits this additional outcome only. It turns out that this is the strongest valid visibility annotation for the contains method. On the one hand, the *weak* visibility, which does not constrain what contains observes, admits the additional unwitnessed outcome of Figure 2, in which contains returns false even after its happens-before predecessor has observed the presence of $\langle 1, 1\rangle$. On the other hand, the *peer* visibility, which requires that contains see the happens-before predecessors of visible operations, is inconsistent with the observed non-atomic outcome, since $\langle 0, 0\rangle$ would be visible too if $\langle 1, 1\rangle$ were.

The non-atomic outcomes of Figures 3 and 4 are due to a distinct optimization: in order to quickly return entry counts without re-scanning all entries, Java's ConcurrentHashMap implementation maintains an entry counter which is kept only loosely in sync with the actual entry count. On the one hand, there is no visibility relaxation which can account for size returning 2, since at no time may the map contain more than one entry. The implementation allows this outcome by effectuating the removal of $\langle 1, 0\rangle$ in the second thread before being interrupted by the insertion of $\langle 1, 1\rangle$, yet decrementing the entry count only after size returns. On the other hand, the *weak* visibility relaxation can account for the non-atomic behavior of isEmpty, since isEmpty need not observe the addition of $\langle 1, 2\rangle$ by its happens-before predecessor. The implementation allows this outcome by effectuating the insertion of $\langle 1, 1\rangle$ in the first thread before being interrupted by the insertion of $\langle 1, 2\rangle$, yet incrementing the entry count only after isEmpty returns; the implementation avoids incrementing the counter for the second entry, since it replaces the existing entry.

While many of Java's concurrent objects methods are non-atomic, the majority can still be given precise and checkable specifications using our proposed visibility-relaxation annotations. We exemplify Java specifications here because Java is mainstream, yet the lack of precise consistency specifications for software APIs is pervasive. This problem is profound since without consistency guarantees client-side reasoning, validation, and sound optimization are all impossible.

3 VISIBILITY RELAXATION

To specify the non-atomic operations of typical software APIs, we propose a simple annotation language which captures the phenomena of limited observation described in Section 2. The semantics of this annotation language is based on an abstract notion of executions, which captures observed return values and ordering constraints among program invocations. Besides considering various linearizations of invocations, our semantics considers which invocations may or may not be observed by any given invocation occurring later in a given linearization. Specifications then dictate the abstract executions admitted by a given abstract data type (ADT) by imposing constraints on this observation relation. Our annotation language supplements the functional specifications of APIs, given by abstract data types, to determine whether the observed behaviors of their implementations shall be admitted.[8]

Without loss of generality — see Remark 1 — we consider a simplistic notion of programs with trivial control and data flow. Formally, an *m-invocation i* is a method name m, along with a vector $\vec{v}$ of argument values, and an identifier capable of distinguishing invocations to the same method with the same argument values; an *M-invocation* is an m-invocation for some method $m \in M$. An *operation* is an invocation paired with a return value. An *abstract data type A* over methods M is a mapping from M-invocation sequences $i_0 i_1 \ldots i_n$ to return-value sequences $A(i_0 i_1 \ldots i_n) = v_0 v_1 \ldots v_n$. A *program $p = \langle po, hbs \rangle$* over an abstract data type A (over methods M) is a partial *program order po* on M-invocations, given as a union of total orders representing threads, along with a set hbs of *happens-before* partial orders on the invocations of po, each including po. Besides program order, these happens-before orders generally represent the platform-defined synchronization constraints imposed externally to API invocations, including the release-to-acquire ordering of monitor actions, and the ordering from write-to-read actions of sequentially-consistent objects.[9] Rather than modeling synchronization actions directly, we capture only their impact via the possible happens-before orders among invocations in a given program.

Throughout this work we write program orders using a familiar notation, as parallel compositions `{...}||{...}` of invocation sequences, e.g., $\{i_1 ; i_2 ; i_3\}$, as in Figures 2–4. The corresponding program order po relates invocations within sequences, e.g., $\langle i_1, i_2 \rangle \in po$, but not invocations of distinct sequences. Our notion of programs supposes that the possible happens-before orders can be determined, and enumerated, statically. While this is not generally possible for programs arbitrary control and data flow, our simple notion of programs enables this static determination for common synchronization primitives like locks and volatile variables — see Remark 1.

Example 3.1. Consider the Java programs listed in Figure 5, in which two threads invoke the add method of a concurrent set. Numbering invocations by their argument values, both programs correspond to the same program order $po = \{\langle 2, 3 \rangle\}$. The left-hand side program imposes happens-before constraints through the use of volatile variables, and there are three possible orders, all statically predictable, depending on the values read by the second thread: either $hb_1 = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 3 \rangle\}$, when $a = 42$, or $hb_2 = \{\langle 1, 3 \rangle, \langle 2, 3 \rangle\}$, when $a = 0$ and $b = 42$, or $hb_3 = \{\langle 2, 3 \rangle\}$, when $a = b = 0$. The corresponding program in our notation is $\langle po, \{hb_1, hb_2, hb_3\} \rangle$. Note that neither Invocation 2 nor 3 may happen before 1, since the first thread does not read synchronization variables. Similarly, the right-hand side program imposes happens-before constraints through the use of locks, and there are two possible orders, both statically predictable, depending on the order of critical sections: either $hb_4 = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 3 \rangle\}$, when the first thread's precedes the second's, or $hb_5 = \{\langle 2, 1 \rangle, \langle 2, 3 \rangle\}$, in the reverse. The corresponding program according to our notation is $\langle po, \{hb_4, hb_5\} \rangle$.

We consider an abstract notion of executions, or behaviors, which captures observed return values and platform-defined happens-before order among invocations in a given program. Formally, an *outcome* of a program $p = \langle po, hbs \rangle$ is a mapping ret from invocations i of p to return values $ret(i)$. An *abstract execution*, or *behavior*, $b = \langle hb, ret \rangle$ of p is a partial happens-before order $hb \in hbs$, along with an outcome ret of p. An *implementation $Impl$* of an abstract data type A maps programs p over A to sets $Impl(p) = \{b_1, b_2, \ldots\}$ of behaviors of p. We refer to the behaviors of $Impl(p)$ as *observed behaviors*.

Remark 1. Our notion of programs includes only trivial control and data flow, and statically-predictable happens-before constraints. For instance, we exclude conditional statements and loops,

[8]The *functional (ADT) specification* of an API determines the permissible return values for API invocation sequences.
[9]Java's *volatile* variables guarantee sequentially-consistent reads and writes.

```
class Main {
    static volatile int v;

    public static void main(String[] args) {
        var s = new ConcurrentSkipListSet();
        new Thread(() -> {
            s.add(1); v = 42;
        }).start();
        new Thread(() -> {
            var a = v; s.add(2); var b = v; s.add(3);
        }).start();
    }
}
```

```
class Main {
    static Lock l = new ReentrantLock();

    public static void main(String[] args) {
        var s = new ConcurrentSkipListSet();
        new Thread(() -> {
            l.lock(); s.add(1); l.unlock();
        }).start();
        new Thread(() -> {
            l.lock(); s.add(2); l.unlock(); s.add(3);
        }).start();
    }
}
```

Fig. 5. Two multithreaded Java programs invoking the add method of a concurrent set object, with three and two statically-predictable happens-before orders, respectively.

as well as passing return values as arguments, in favor of straight-line code with literal argument values. Nevertheless, this simple notion is expressive enough to capture any possible behavior, according to the aforementioned notion of behavior, of programs with arbitrarily complex control flow, data flow, and synchronization. To see this, consider the behavior $b = \langle hb, ret \rangle$ admitted by some execution of a program with arbitrarily complex control and data flow. The simple program order po in which each thread invokes the operations executed by one unique thread of the original complex program, passing as literals the argument values to those executed invocations, is guaranteed to admit the exact same outcome ret. Combined with the original behavior b's happens-before order hb, the program $\langle po, \{hb\} \rangle$ is guaranteed to admit b.

Our annotation language semantics relies on a few basic notions of sequences and orders. We denote the *prefix* of a sequence σ up to and including an element α by $\sigma(\alpha)$, and the *prefix* of a partial order π up to and including α by $\pi(\alpha) = \{\langle \alpha_1, \alpha_2 \rangle \in \pi : \langle \alpha_2, \alpha \rangle \in \pi \text{ or } \alpha_2 = \alpha\}$. A sequence σ_1 is called a *subsequence* of another sequence σ_2, denoted by $\sigma_1 \preceq \sigma_2$, when σ_1 is obtained from σ_2 by deleting elements. We extend the notion of subsequence to partial orders and say that an order π_1 is a *suborder* of another order π_2 if $\pi_1 \subseteq \pi_2$. For uniformity, we write $\pi_1 \preceq \pi_2$ when π_1 is a suborder of π_2. Also, for simplicity, we don't make the distinction between sequences and total orders, reusing notions like prefix and subsequence in the context of total orders. Finally, we write $i \in \sigma$ when i is an element of the sequence σ and $i \in \pi$ when i is an element of the partial order π, i.e., there exists j such that $\langle i, j \rangle$ or $\langle j, i \rangle$ is included in π.

Extending the usual notion to include invocations' observations, a *linearization* $\ell = \langle lin, vis \rangle$ of the behavior $b = \langle hb, ret \rangle$ is:

- a total order lin, which includes hb,[10] and
- a function vis, mapping each invocation i to a subsequence of $lin(i)$ including i.

We say an invocation i_1 is *visible* to i_2 when $i_1 \in vis(i_2)$, and that i_2 *sees* i_1. The linearization ℓ is *admitted* by an abstract data type A when for each invocation i of ℓ, the outcome $ret' = A(vis(i))$ of the invocations visible to i is consistent with ret on i's return value, i.e., $ret(i) = ret'(i)$ — the outcomes ret and ret' may disagree on return values for the other invocations visible to i. The behavior b is *admitted* by A when some linearization of b is, and an outcome ret is *admitted* by A when some behavior with outcome ret is.

[10]Unlike *linearizability* [Herlihy and Wing 1990], which considers linearizations of the dynamic real-time order among invocations, we consider linearizations of the weaker dynamic happens-before order, i.e., a superset. This distinction is orthogonal to the relaxation of visibilities, but enables validation. Section 4 considers this distinction in depth.

$$
\begin{aligned}
\text{weak}(i,\ell,b) &\Leftrightarrow \mathit{true} \\
\text{basic}(i,\ell,b) &\Leftrightarrow \mathit{hb}(i) \preceq \mathit{vis}(i) \\
\text{monotonic}(i,\ell,b) &\Leftrightarrow \forall j \in \mathit{hb}(i).\ \mathit{vis}(j) \preceq \mathit{vis}(i) \\
\text{peer}(i,\ell,b) &\Leftrightarrow \forall j \in \mathit{vis}(i).\ \mathit{hb}(j) \preceq \mathit{vis}(i) \\
&\qquad \wedge \text{monotonic}(i,\ell,b) \\
\text{causal}(i,\ell,b) &\Leftrightarrow \forall j \in \mathit{vis}(i).\ \mathit{vis}(j) \preceq \mathit{vis}(i) \\
&\qquad \wedge \text{basic}(i,\ell,b) \\
\text{complete}(i,\ell,b) &\Leftrightarrow \mathit{vis}(i) = \mathit{lin}(i)
\end{aligned}
$$

Fig. 6. The semantics of visibility predicates, given a linearization $\ell = \langle \mathit{lin}, \mathit{vis} \rangle$ of a behavior $b = \langle \mathit{hb}, \_ \rangle$ of program $p = \langle \mathit{po}, \_ \rangle$.

```
function expected({ po, hbs }, Impl, S) {
  for (let hb of hbs) {
    for (let { lin, vis } of hb.lins()) {
      if (!S.isSatisfied(lin, vis))
        continue;
      let ret = {};
      for (let i of lin) {
        let seq = vis(i);
        let res = Impl.execute(seq);
        ret[i] = res[res.length - 1];
      }
      yield { hb, ret };
} } }
```

Fig. 7. Computing the expected behaviors for a given program and specification, based on the sequential executions of a given implementation.

We define our annotation language around a small set of predicates which constrain the observations made by a given invocation. Formally, a *visibility predicate* $\varphi(i, \ell, b)$ is a first-order predicate on invocations i in linearizations ℓ of behaviors b. Figure 6 defines the semantics of six increasingly stronger visibility predicates: weak, basic, monotonic, peer, causal, and complete. Figure 1 illustrates this semantics graphically, besides for weak and complete, since their constraints are independent from the visibilities of other methods.

The predicates of our annotation language are adapted from *happens-before consistency* [Manson et al. 2005], *sequential consistency* [Lamport 1979], and concepts from distributed systems, including *causal consistency* [Lamport 1978] and *session guarantees* [Terry et al. 1994]. Both causal consistency and the causal predicate require that visibilities be transitive, though our notion is at the per-method granularity rather than per-API granularity. Sequential consistency (and linearizability – see Section 4) is related to the complete predicate: both require that invocations observe their predecessors in linearization order. The basic predicate requires invocations to see their happens-before predecessors, which coincides, e.g., with the wording of the Java Language Specification: "if one action happens-before another, then the first is visible and ordered before the second."

More subtle is the connection to session guarantees for replicated databases [Terry et al. 1994]. In that context, a session is an abstraction for the sequence of database operations performed during the execution of a program. The basic, monotonic, and peer predicates correspond to guarantees on database read and update operations. Specifically:

- our *basic* predicate corresponds to a strengthening (from program order to happens-before order) of the *read-your-writes guarantee*, which states that the updates made within a session are visible to reads within that session;
- our *monotonic* predicate corresponds to the *monotonic reads guarantee*, which states that reads observe an increasingly up-to-date state of the database, i.e., increasing sets of updates;
- our *peer* predicate corresponds to the *monotonic writes guarantee*, which states that a read can observe an update only if it has observed all the preceding updates from its session, in our case, all the other updates that happen-before that update.

Besides the fact that our notions are at the per-method granularity rather than per-API, these notions differ as we define predicates to satisfy the following linear order to simplify specification.

Lemma 3.2. *For each invocation i in a linearization ℓ of behavior b:*

$$\begin{aligned}&\mathsf{complete}(i,\ell,b) \Rightarrow \mathsf{causal}(i,\ell,b) \Rightarrow \mathsf{peer}(i,\ell,b)\\ &\Rightarrow \mathsf{monotonic}(i,\ell,b) \Rightarrow \mathsf{basic}(i,\ell,b) \Rightarrow \mathsf{weak}(i,\ell,b).\end{aligned}$$

Our specification methodology annotates each method with a visibility predicate according to the semantics above. To map visibility predicate semantics to method annotations, we define $\varphi(m,\ell,b)$ iff $\varphi(i,\ell,b)$ for all m-invocations i in linearization ℓ of behavior b.

Definition 3.3. A *visibility specification S* of an abstract data type over methods M is a mapping from methods $m \in M$ to visibility predicates $S(m)$.

3.1 Client-Side Reasoning

While we focus here on characterizing the relaxations admitted by existing implementations, client-side reasoning using relaxed-visibility specifications is also an important consideration. One possible direction could be to extend existing methodologies for linearizable objects, wherein programmers reason about sequences of ADT invocations using ADT specifications only, without implementation details. To extend this methodology to relaxed visibility, the programmer could also consider the possible visibilities among invocations in a given sequence; an operation's return value is determined by the subsequence of its visible invocations, maintaining the utility of completely-sequential ADT specifications. We intend to explore such client-side reasoning in future work.

3.2 Validating Implementations

To characterize the relaxations admitted by existing implementations, we establish a heterogeneous notion of validity, allowing methods to exercise distinct visibilities, along with a practical validation procedure. A linearization ℓ of a behavior b of a program p over abstract data type A *satisfies* a visibility specification S when $\varphi(m,\ell,b)$ holds for each $m \in \mathrm{dom}(S)$ and $\varphi = S(m)$. Then b *satisfies* S when some linearization ℓ does, and is *expected* when that ℓ is admitted by A. An implementation *Impl* of A *satisfies* S when each observed behavior is expected, for every program over A.

Definition 3.4. The *visibility-specification validity problem* is to determine whether a given implementation *Impl* satisfies a given visibility specification S.

In this work we reduce the validity problem to assertion checking. For a given program and specification, we compute the set of expected behaviors, and add instrumentation to the program to record an observed behavior, along with a single assertion ensuring that the observed behavior is among the expected behaviors. This assertion fails whenever the given program admits a behavior inconsistent with the specification. The algorithm of Figure 7 demonstrates the calculation of expected behaviors using the sequential executions of the given implementation as a stand-in for its abstract data type. This substitution is useful in practice, since the abstract data type of a given implementation is typically not specified formally. We assume that the method `hb.lins` returns the linearizations of its target happens-before order, the method `S.isSatisfied` returns true iff its argument linearization satisfies its target visibility specification, and the method `Impls.execute` returns the sequence of values returned by its argument invocation sequence.

Remark 2. Besides determining the validity of operations' return values, API specifications can contribute to happens-before, e.g., the insertion of a given element in a Java collection happens-before its corresponding retrieval. While we do not address this aspect of API specification directly in this work, our validation methodology applies equally well on under the following assumption: any given sequence of ADT operations cannot yield a happens-before constraint that relates an invocation to some predecessor. Under this assumption, any happens-before constraints contributed

Table 1. Computing expected behaviors for the program in Figure 2.

ConcurrentHashMap: contains
`{ put(1,0); contains(0) } || { put(0,0); put(1,1) }`

linearization	visibility of contains	outcome
`put(1,0); contains(0); put(0,0); put(1,1)`	`put(1,0)`	`null, true, null, 0`
`put(1,0); put(0,0); contains(0); put(1,1)`	`put(1,0)`	`null, true, null, 0`
	`put(1,0); put(0,0)`	`null, true, null, 0`
`put(1,0); put(0,0); put(1,1); contains(0)`	`put(1,0)`	`null, true, null, 0`
	`put(1,0); put(0,0)`	`null, true, null, 0`
	`put(1,0); put(1,1)`	`null, false, null, 0`
	`put(1,0); put(0,0); put(1,1)`	`null, true, null, 0`
`put(0,0); put(1,1); put(1,0); contains(0)`	`put(0,0); put(1,1); put(1,0)`	`1, true, null, null`
`put(0,0); put(1,0); put(1,1); contains(0)`	`put(0,0); put(1,0)`	`null, true, null, 0`
	`put(0,0); put(1,0); put(1,1)`	`null, true, null, 0`
`put(0,0); put(1,0); contains(0); put(1,1)`	`put(0,0); put(1,0)`	`null, true, null, 0`

within a given linearization is guaranteed to be consistent with that linearization. This implies that the behaviors computed in Figure 7, augmented with the contributed happens-before constraints, are valid, and precisely characterize the behaviors of multi-object programs as well.

The computation of expected behaviors for the program in Figure 2, with respect to a visibility specification where contains has monotonic visibility and the rest of the methods have complete visibility, is given in Table 1. The enumeration of linearizations consistent with happens-before order is nested with an enumeration of visibilities for contains — the remaining invocations' visibilities being fixed — where contains sees at least as many invocations as its predecessor put(1,0) in program order, by the monotonicity assumption. This enumeration yields the three observed outcomes of Figure 2, which coincide exactly with three expected behaviors, given the unique happens-before order which is equal to program order, due to the lack of additional synchronization.

4 SEQUENTIAL HAPPENS-BEFORE CONSISTENCY

Unlike the familiar notion of atomicity formalized by *linearizability* [Herlihy and Wing 1990], the visibility specifications described in Section 3 are agnostic to the real-time order among invocations in an execution.[11] Instead of real-time order, our notion of linearization is only required to include the platform-defined happens-before order among invocations. By ignoring real-time order, our notion is more akin to the linearizations of *sequential consistency* [Lamport 1978], which are only required to include program order. Requiring linearizations to include the stronger happens-before order, instead of program order alone, provides a stronger criterion which is more aligned with the specifications of modern platforms like Java and C++. Such platforms provide happens-before ordering guarantees among the actions of shared-memory synchronization objects without posing constraints on the real-time order in which unrelated actions execute. These guarantees permit high-level program reasoning even amidst drastic low-level program optimizations. To reflect our alignment with the happens-before order as defined by platform specifications, specialized to objects with sequential specifications,[12] we refer to the corresponding criterion as *(relaxed-visibility) sequential happens-before consistency.*

[11]Two invocations are *real-time ordered* when the first invocation's return event is executed before the second's call event.
[12]In this work we consider objects whose *concurrent* behaviors are explained as the results of linear *sequences* of invocations.

In this section we compare the linearizability criterion with sequential happens-before consistency (SHBC).[13] Since the relaxation of visibilities within a given linearization is an orthogonal concern, for the remainder of this section we assume only visibility specifications in which every method is annotated as complete, and thus consider only linearizations in which every invocation sees all previously-linearized invocations. We argue that while linearizability is generally stronger than SHBC, its direct monitoring is impractical on modern platforms like Java and C++ whose specifications do not expose the real-time order among invocations directly. Instead, these platform specifications expose only a happens-before order, which is not generally implied by real-time order. Thus, according to platform specifications, SHBC is in some sense the strongest criterion comparable to linearizability for which runtime verification is possible. Nevertheless, despite the disparity between linearizability and SHBC per execution, surfacing when real-time orderings are not exposed by happens-before, we demonstrate that with varying platform assumptions, linearizability and SHBC are equivalent to varying degrees, e.g., in the limit over all executions of all programs invoking a given implementation. The remainder of this section formalizes these arguments, and provides a general framework under which linearization-based criteria can be adapted to modern platforms like Java and C++.

We recall the notion of *abstract execution* of an implementation *Impl* of an abstract data type A and program p over A from Section 3, i.e., a *behavior* $b = \langle hb, ret \rangle$ recording the happens-before order hb and outcomes *ret* among invocations of p. For the purposes of this section we consider an analogous notion of *concrete execution* $e = \langle rt, ret \rangle$ which records the real-time order rt among invocations. Furthermore, we identify the executions of p with *Impl* as pairs $Impl(p) = \{\langle b_1, e_1 \rangle, \langle b_2, e_2 \rangle, \ldots\}$ of abstract and concrete executions as being recorded simultaneously from some *physical execution* of a given platform, requiring the outcomes of b_i and e_i to be identical. We say that an execution $\langle b, e \rangle$ is *sequentially happens-before consistent (SHBC)* or *linearizable*, respectively, with a given abstract data type A when there exists some linearization $\ell = \langle lin, vis \rangle$ of b or e, respectively, admitted by A such that $vis(i) = lin(i)$ for all invocations i, and *lin* includes *hb* or *rt*, respectively. Intuitively, an execution is linearizable if each individual invocation appears to take effect instantaneously at some point between its call and return actions, hence the relevance of real-time order. We say that the implementation *Impl* of a given abstract data type A satisfies criteria χ, e.g., linearizability, when all executions of $Impl(p)$ satisfy χ, for all programs p over A.

Formally, we characterize a *platform* by the set of implementations that can be realized thereon, implicitly quantifying implementations with respect to a given platform. We say that a platform is *real-time sound* (RTS) when happens-before order implies real-time order: if $\langle i, j \rangle \in hb$ then $\langle i, j \rangle \in rt$, for all invocations i and j of all executions $\langle b, e \rangle \in Impl(p)$ with $b = \langle hb, \_ \rangle$ and $e = \langle rt, \_ \rangle$, for all implementations *Impl* of abstract data type A and programs p over A. RTS justifies SHBC as complete yet potentially-unsound approximation to linearizability.

Theorem 4.1. *Linearizable executions are SHBC on RTS platforms.*

Proof. An execution $\langle \langle hb, ret \rangle, \langle rt, ret \rangle \rangle$ being linearizable implies the existence of a linearization $\ell = \langle lin, vis \rangle$ admitted by A where $rt \subseteq lin$ and $vis(i) = lin(i)$ for all invocations i. The RTS property implies that $hb \subseteq rt$, which is enough to conclude that $hb \subseteq lin$, hence $\langle b, e \rangle$ is SHBC. □

Corollary 4.2. *Linearizability implies SHBC on RTS platforms.*

Real-time soundness captures the intent of platform-defined happens-before orders: even when the invocations of synchronization primitives overlap, when one happens-before another, the invocations preceding the first are generally guaranteed to complete before those succeeding the second begin. For the remainder of this section, we assume that all platforms are real-time sound.

[13] Here we focus on enforcement, as opposed to other considerations like compositionality [Dongol et al. 2018]; see §7.

To state a corresponding notion of completeness, we must leverage platform specifications to ensure the manifestation of ordering constraints. We say that a function θ is a *synchronization transformation* when for any program $p = \langle po, hbs \rangle$ the program $\theta(p) = \langle po, hbs' \rangle$ has identical invocations, yet a possibly distinct set of happens-before orders. Then we say that a platform is *real-time complete* (RTC) when there exists some synchronization transformation θ such that for all implementations *Impl* of an abstract data type A and programs p over A:

- $\theta(p)$ preserves the concrete executions of p: for any execution $\langle \_, e \rangle$ of $Impl(p)$ there exists an execution $\langle \_, e' \rangle$ of $Impl(\theta(p))$ with $e = e'$, and
- real-time order in $\theta(p)$ implies happens-before: if $\langle i, j \rangle \in rt$ then $\langle i, j \rangle \in hb$ for all invocations i and j of any execution $\langle b, e \rangle \in Impl(\theta(p))$ with $b = \langle hb, \_ \rangle$ and $e = \langle rt, \_ \rangle$.

We refer to transformations θ meeting the properties above as *real-time instrumentations*.

These properties are not generally guaranteed by modern platforms like Java and C++. In particular, the first property is rather demanding: it essentially stipulates the possibility of adding synchronization which does not interfere with existing operations. While modern platform specifications do not contradict this property, it is unlikely to be met on typical platform implementations since, e.g., memory barriers typically synchronize the accesses to *all* memory locations, rather than *individual* memory locations independently. The second property, also not contradicted by modern platform specifications, is more likely to be realizable. It essentially stipulates the ability to treat invocation call and return actions as read-acquire and write-release actions, respectively. Note that this ability is of little value without the first property, since interference among these actions and with existing operations would render operations directly atomic, thus masking possible inconsistent behaviors of the unmonitored program.

Example 4.3. Typical software platforms generally provide two possible approaches for real-time instrumentation. On the one hand, the *memory-based approach*, as exemplified by Figure 8, uses auxiliary shared variables (`vs1`, `vs2`) that are read from and written to, respectively, immediately before and after each ADT invocation, in order to record the happens-before order (`hbs`). To be a valid real-time instrumentation, this approach requires at least three platform guarantees. First, the platform must not reorder accesses to the shared variables (`vs1`, `vs2`); while this could easily be achieved by marking them as volatile, this would almost certainly cause interference, and mask possible inconsistent behaviors. Second, the platform must not allow delays, e.g., context switching, between shared-variable reads and invocation call actions, nor between return actions and the corresponding writes. Finally, the platform must guarantee independence between the writes to distinct memory locations, in order to prevent interference from the shared variable accesses. This ability is akin to the partial-store ordering (PSO) memory model [SPARC International 1994], rather than the total-store ordering (TSO) model [Sewell et al. 2010], in which writes to all locations share the same write-back buffer, and thus interfere. While certain platform implementations may provide some of these guarantees, modern platform specifications generally guarantee neither.

On the other hand, the *clock-based approach*, as exemplified by Figure 9, uses system clock values (read into `ts1` and `ts2`) before and after each ADT invocation. Compared to the memory-based approach, the clock-based approach is less likely to suffer from memory-access interference, but still requires that the platform does not allow delays, e.g., context switching, between clock accesses and invocation call or return actions. However, to be a valid real-time instrumentation, this approach also imposes at least two other platform requirements. First, the platform must guarantee sufficiently high-precision such that two distinct accesses read two distinct values; otherwise actual real-time orderings will be unobservable. Second, the platform must guarantee negligible latency compared to invocations' call and return actions; otherwise the clock-value accesses can themselves interfere with the sensitive timing constraints under which certain executions are admitted. Again,

```
class Main {
  static boolean vs1[] = { false };
  static boolean vs2[] = { false, false };
  static boolean hbs[][][] = new boolean[2][][];

  public static void main(String[] args) {
    Set s = new ConcurrentSkipListSet();
    var t1 = new Thread(() -> {
      boolean[] hb1 = { vs2[0], vs2[1] }; s.add(1);
      vs1[0] = true;
      boolean[][] hb = { hb1 }; hbs[0] = hb;

    });
    var t2 = new Thread(() -> {
      boolean[] hb1 = { vs1[0] }; s.add(2);
      vs2[0] = true;
      boolean[] hb2 = { vs1[0] }; s.add(3);
      vs2[1] = true;
      boolean[][] hb = { hb1, hb2 };  hbs[1] = hb;


    });

    t1.start(); t2.start(); t1.join(); t2.join();
    // happens before is recorded in hbs[][][]
} }
```

Fig. 8. Memory-based real-time instrumentation.

```
class Main {
  static long ts1[] = { Long.MAX_VALUE };
  static long ts2[] = { Long.MAX_VALUE, Long.MAX_VALUE };
  static boolean hbs[][][] = new boolean[2][][];

  public static void main(String[] args) {
    Set s = new ConcurrentSkipListSet();
    var t1 = new Thread(() -> {
      var t1 = System.nanoTime(); s.add(1);
      ts1[0] = System.nanoTime();
      boolean[] hb1 = { ts2[0]-t1 < 0, ts2[1]-t1 < 0 };
      boolean[][] hb = { hb1 }; hbs[0] = hb;
    });
    var t2 = new Thread(() -> {
      var t1 = System.nanoTime(); s.add(2);
      ts2[0] = System.nanoTime();
      var t2 = System.nanoTime(); s.add(3);
      ts2[1] = System.nanoTime();
      boolean[] hb1 = { ts1[0] - t1 < 0 };
      boolean[] hb2 = { ts1[0] - t2 < 0 };
      boolean[][] hb = { hb1, hb2 };  hbs[1] = hb;
    });

    t1.start(); t2.start(); t1.join(); t2.join();
    // happens before is recorded in hbs[][][]
} }
```

Fig. 9. Clock-based real-time instrumentation.

while certain platform implementations may provide some of these guarantees, we find that modern platform specifications do not generally guarantee all of them together.

Since the definition of real-time completeness requires the addition of program synchronization to witness real-time constraints, we do not compare linearizability to SHBC directly on arbitrary programs. Instead, we consider a notion of equivalence which considers programs instrumented with additional synchronization. For a given synchronization transformation θ, we say that a program $\theta(p)$ is *instrumented*. We say that two criteria χ_1 and χ_2 are *equivalent* when for every implementation *Impl* of a given abstract data type *A*, *Impl* satisfies χ_1 iff *Impl* satisfies χ_2. In the case of RTC platforms, every execution of an instrumented program captures real-time order precisely.

Theorem 4.4. *SHBC executions of instrumented programs are linearizable on RTC platforms.*

Proof. Let $\langle\langle hb, ret\rangle, \langle rt, ret\rangle\rangle$ be an SHBC execution of an instrumented program $\theta(p)$. Then, there exists a linearization $\ell = \langle lin, vis\rangle$ admitted by A where $hb \subseteq lin$ and $vis(i) = lin(i)$ for all invocations i. By RTC, we have $rt \subseteq hb$, implying $rt \subseteq lin$, and that the execution is linearizable. □

Since there exists an instrumented program capturing the concrete executions of every non-instrumented program precisely on RTC platforms, in the limit over all programs invoking the methods of a given ADT, linearizability is equivalent to SHBC.

Corollary 4.5. *SHBC is equivalent to linearizability on RTC platforms.*

Proof. By Corollary 4.2, if an implementation is linearizable, then it is SHBC. For the reverse, let *Impl* be an SHBC implementation, and assume by contradiction that there is an execution $\langle\langle hb, ret\rangle, \langle rt, ret\rangle\rangle$ of a program p which is not linearizable. By the RTC property, there exists an execution $\langle\langle hb', ret\rangle, \langle rt, ret\rangle\rangle$ of $\theta(p)$ where $rt \subseteq hb'$. Since the original execution is not linearizable, we have that there exists no linearization $\ell = \langle lin, vis\rangle$ admitted by A where $rt \subseteq lin$ and $vis(i) =$

$lin(i)$ for all invocations i. Since $rt \subseteq hb'$, we get that the latter execution is not SHBC which contradicts the hypothesis. □

Real-time completeness corresponds to a fairly demanding assumption on the platforms which programs execute, but it is not the weakest assumption for which SHBC and linearizability coincide. In particular, the first requirement, which demands that the synchronization transformation θ preserves real-time executions can be relaxed, to demand only that the happens-before order of some other execution reflects the original real-time orders. Formally, we say that a platform is *real-time limit complete* (RTLC) when there exists some synchronization transformation θ such that for all implementations *Impl* of an abstract data type A and programs p over A:

- $\theta(p)$ reflects the concrete executions of p as abstract executions: for any execution $\langle \_, e\rangle$ of $Impl(p)$ there exists an execution $\langle b, \_\rangle$ of $Impl(\theta(p))$ such that $rt = hb$ and $ret = ret'$, where $e = \langle rt, ret\rangle$ and $b = \langle hb, ret'\rangle$.

Individual SHBC executions of instrumented programs on RTLC platforms are not guaranteed to be linearizable, since there is no guarantee that the real-time order of a given execution is reflected in the happens-before order of the same execution. Nevertheless, in the limit over all executions, the real-time order of every concrete execution is guaranteed to be reflected by the happens-before order of some abstract execution. Thus the same notion of equivalence for RTC platforms ultimately applies to RTLC platforms.

Theorem 4.6. *SHBC is equivalent to linearizability on RTLC platforms.*

Proof. By Corollary 4.2, linearizable implementations are SHBC. For the reverse, let *Impl* be an SHBC implementation, and assume by contradiction that there is an execution $\langle\langle hb, ret\rangle, \langle rt, ret\rangle\rangle$ of a program p which is not linearizable. By the RTLC property, there exists an execution $\langle\langle hb, ret\rangle, \_\rangle$ of $\theta(p)$ where $hb = rt$. Since the original execution is not linearizable, we have that there exists no linearization $\ell = \langle lin, vis\rangle$ admitted by A where $rt \subseteq lin$ and $vis(i) = lin(i)$ for all invocations i. Since $hb = rt$, we get that the latter execution is not SHBC which contradicts the hypothesis. □

5 EMPIRICAL RESULTS

To assess the value of visibility relaxation and sequential happens-before consistency (SHBC) as practical specification and validation methodologies, we have studied their application to Java concurrent objects.[14] Our evaluation aims to verify the following hypotheses:

(1) Java's concurrent object methods are not generally atomic.[15]
(2) The majority of these methods have consistent visibility relaxations.
(3) SHBC-based validation is effective in uncovering consistency violations.

While the first hypothesis could be deduced from the second, by showing maximality for the relaxed-visibility consistencies, we include it to emphasize an unheralded fact: consistency relaxation is hidden in plain sight. The architects of the most-mainstream programming languages willfully compromise consistency for efficiency in widely-used concurrent object implementations.

We validate these hypotheses in turn in Sections 5.1–5.3 using an open-source and publicly-available prototype implementation of SHBC-based consistency checking, which we developed for this purpose.[16] Our prototype takes the API specification of a single Java class as input, including a signature and (complete, causal, peer, monotonic, basic, or weak) visibility annotation for each

[14]Our study is based on the Java SE Development Kit 9.
[15]While the term *atomic* for concurrent objects is generally understood as *linearizable*, here we equate the term with SHBC. Since SHBC violations are also linearizability violations — see Section 4 — our reported consistency violations are valid for both interpretations of the term.
[16]The Violat consistency-checking tool: https://github.com/michael-emmi/violat

method, and operates in one of two modes. The *single-program mode* takes a *program* invoking methods of the given class as an additional input, while the *automatic mode* generates a sequence of programs automatically. Programs are expressed as sequences of threads, each thread invoking a sequence of methods, without any other statements nor loss of generality. Our prototype identifies SHBC-violations of the given class's consistency specification in executions of the given program(s). Section 5.4 discusses the efficacy of validation based on testing.

In automatic mode, our prototype generates a bounded sequence of programs at random, given minimum and maximum bounds on the number of threads, invocations, and argument values. By default, we generate programs with 2 threads, from 3 to 6 invocations, and up to 2 values. Since Java's concurrent objects are essentially generic collection types, argument value types are essentially unconstrained. For simplicity we suppose that argument values are either integers from $0 \dots n$, collections of integers, or integer-to-integer maps. We leave the handling of higher-order argument values to future work. To avoid an excessive enumeration of programs which invoke only read-only methods, which are unlikely to expose consistency violations, or invoke only non-atomic methods, which impedes the diagnosis of consistency violations, we consider a weighted random selection of invocation targets: atomic mutators are 3 times as likely to be selected as non-atomic or read-only methods; this distribution is somewhat arbitrary, but guided by experience.

To simplify our evaluation, we consider only programs without synchronization between threads — beyond that which is already present in the implementation of API methods. This is generally unsound, since, in principle, the only inconsistent behaviors admitted by a given implementation may have happens-before orders stronger than program order. However, as discussed in Section 5.3, our empirical study demonstrates that a vast number of inconsistent behaviors are admitted with program order alone, and so we have left it to future work to generate and test programs with additional synchronization. This restriction yields a notable optimization to our testing methodology: rather than enumerating and recording entire program behaviors, we can reason about program outcomes only, significantly simplifying both enumeration and recording. Accordingly, for the remainder of this section we reduce the scope of consistency from behaviors to outcomes. The extension to behaviors is straightforward, since behaviors can be reconstituted from the outcomes of programs with additional volatile variable accesses, e.g., similarly to the memory-based instrumentation of Section 4 — the key difference being that interference in this case is *intended* when witnessing happens-before orderings, i.e., as opposed to witnessing real-time orderings.

To validate the executions of individual programs, our prototype leverages an off-the-shelf stress testing tool,[17] subjecting each program to 1 second of stress testing; this default setting is sufficient in our experience. We construct tests according to Section 3, by enumerating the expected outcomes, i.e., the invocation results consistent with the relaxed visibilities of the API specification. For each tested execution, the stress testing framework checks that invocations' return values collectively match one of the expected outcomes. Unexpected outcomes indicate consistency violations, which our prototype reports, along with their corresponding program.

Our empirical evaluation covers 100K randomly generated programs for each of 7 out of 14 classes from Java's concurrent object package.[18] Those excluded either did not exhibit atomicity violations, or are not characterized by sequential specifications.[19] We also exclude sequentially-nondeterministic methods like hashCode,[20] bulk operations,[21] iterators, higher-order functions, and methods with mutable output parameters, all of which require orthogonal extensions.

[17]The Java Concurrency Stress tests (jcstress): http://openjdk.java.net/projects/code-tools/jcstress/.
[18]The generation and testing of 100K programs takes roughly 10 hours on a quad-core 4Ghz Intel Core i7 iMac.
[19]Classes like SynchronousQueue rely on inter-thread interaction, and cannot be characterized by sequential behaviors.
[20]The default hashCode implementation depends on objects' memory addresses.
[21]Bulk operations include putAll, addAll, containsAll, removeAll, retainAll.

5.1 Non-Atomicity of Java Concurrent Object Methods

Our initial finding is that Java's concurrent object methods are not generally atomic. Table 2 witnesses the non-atomicity of over 50 methods across the classes evaluated. Each row lists the violation of the given visibility level, for the given method, invoked in the given program, as witnessed by the given outcome. Since a violation to any of the visibility levels is also a violation of atomicity, none of the listed methods are atomic. In order to attribute a given violation to the given method under test, we assume a small number of atomic methods per class. Specifically, we assume the atomicity of sets' add, remove, and contains methods, of maps' put, get, remove, and containsKey methods, and queues' offer, poll, and peek methods. This assumption is reasonable in our experience, since these basic methods are heavily scrutinized by both researchers and practitioners.

While non-atomicity is somewhat unsurprising for some methods, since their official API specifications refer to weakened consistency, the specifications of several methods lack any mention of relaxation. For instance, the getLast, peekLast, pollLast, removeLast, addFirst, and offerFirst methods of the ConcurrentLinkedDeque are surprisingly non-atomic, while their getFirst, peekFirst, pollFirst, removeFirst, addLast, and offerLast methods do behave atomically. We suspect that many or all of these methods are expected to be atomic, since further investigation reveals the subtle omission of Java's final keyword attached to certain local variables initialized from memory, which has implications on the admission of weak-memory effects.

Consequently, some of these violations correspond to existing bug reports: the clear method of ConcurrentSkipListMap and ConcurrentSkipListSet,[22] and the addFirst, peekLast, and pollLast methods of ConcurrentLinkedDeque;[23] the latter report was submitted by the authors of this work.[24] These non-atomic behaviors were acknowledged as bugs and patched in upcoming versions of Java. These admissions underscore the need for precise specification and validation methodologies like ours, without which such bugs are difficult to identify.

5.2 Relaxed Visibility of Java Concurrent Object Methods

Our second finding is that the majority of Java's concurrent object methods are consistent with relaxed-visibility specifications.[25] Table 3 lists the strongest consistent specifications for the classes evaluated, in the sense that escalating any of the given methods' visibility annotations is inconsistent. Table 2 substantiates the strength of each method's annotation by listing observed violations to stronger annotations. For example, we annotate ConcurrentHashMap's contains method with monotonic visibility because it is consistent with all test programs, yet peer visibility is inconsistent with some test programs, e.g., the program listed in Table 2.

Our methodology for deriving specifications is amenable to automation, and works by considering the annotation for each weakly-consistent method m individually. As noted in Section 5.1, we assume a small set M of atomic methods per class, e.g., sets' add, remove, and contains methods. Starting with the complete visibility for m, we enumerate random test programs invoking $M \cup \{m\}$. Upon discovering a violation we weaken m's visibility, e.g., from complete to causal, until either we discover violations even with weak visibility, in which case m cannot be given an annotation, or we obtain sufficient confidence in the current annotation. Unsoundness due to eager termination of this step can be caught in a subsequent step, after individual methods' annotations have been derived, in which we enumerate random test programs invoking any methods which carry annotations. While the overall process is also unsound, since we may never generate certain programs expressing a

[22]ConcurrentSkipListSet.clear() can leave the set in an invalid state: https://bugs.openjdk.java.net/browse/JDK-8166507.

[23]ConcurrentLinkedDeque linearizability: http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8188900.

[24]The bug report does not reveal the authors' name.

[25]We tested each implementation on 100K programs over roughly 10 hours on a quad-core 4Ghz Intel Core i7 iMac.

Table 2. Non-atomic outcomes in Java collections. We list return values in program-text order, and abbreviate null (N), true (T), false (F), exception (E), map entries (*key*=*val*), arrays/lists ([*elems*]), and sets/maps ({*elems*}).

program / **method**	outcome	violated visibility	frequency
ConcurrentHashMap			
{put(0,0); put(1,1); put(1,1)} || {put(0,1); **clear**()}	N,N,N,N,**()**	weak	1 / 2,845,260
{put(0,0);remove(1)} || {put(1,0);**contains**(0)}	N,0,N,**F**	peer	6 / 1,508,770
{get(1);**containsValue**(1)} || {put(1,1);put(0,1);put(1,0)}	1,**F**,N,N,1	peer	1 / 3,993,110
{put(0,1);put(1,0)} || {**elements**()}	N,N,**[0]**	peer	3 / 1,665,650
{put(0,1);put(1,0)} || {**entrySet**()}	N,N,**[1=0]**	peer	23 / 2,688,890
{ put(1,1) } || { put(1,2); **isEmpty**() }	N,1,**T**	basic	57 / 4,136,690
{put(0,1);put(1,1)} || {**keySet**()}	N,N,**[1]**	peer	18 / 5,048,060
{**keys**()} || {put(0,1);put(1,1)}	**[1]**,N,N	peer	13 / 1,721,300
{put(1,0); put(1,1); **mappingCount**()} || {remove(1)}	N,N,**2**,0	weak	52 / 2,231,190
{put(1,0); put(1,1); **size**()} || {remove(1)}	N,N,**2**,0	weak	57 / 2,659,700
{put(0,1);put(1,1)} || {**toString**()}	N,N,**1=1**	peer	120 / 3,948,560
{put(0,1);put(1,0)} || {**values**()}	N,N,**[0]**	peer	99 / 2,836,280
ConcurrentSkipListMap			
{put(1,1);**clear**();remove(1);containsKey(1)} || {get(1);put(0,0)}	N,N,N,T,1,N	weak	8 / 3,296,430
{put(0,0);remove(1)} || {put(1,0);**containsValue**(0)}	N,0,N,F	peer	2 / 3,418,150
{put(0,1);**entrySet**()} || {put(0,0);put(1,1)}	N,[0=1,1=1],1,N	peer	148 / 1,271,070
{put(0,1);put(1,0)} || {put(1,1);**pollFirstEntry**();get(1);remove(0)}	N,1,N,1=0,N,1	weak	1 / 1,532,650
{get(0);put(1,0);put(0,0)} || {put(0,1);**pollLastEntry**();put(0,1)}	N,N,1,N,0=0,N	weak	5 / 1,513,130
{put(0,1);remove(1)} || {put(1,1);**size**()}	N,1,N,0	peer	1 / 3,898,810
{put(0,1);containsKey(0);put(0,0);put(1,0)} || {**tailMap**(0)}	N,T,1,N,{0=1,1=0}	peer	56 / 2,291,450
{put(0,1);put(1,1)} || {put(0,0);**toString**()}	0,N,N,[0=0,1=1]	peer	8 / 2,269,590
{put(0,1);put(1,1)} || {put(0,0);**values**()}	0,N,N,{0,1}	peer	12 / 2,069,060
ConcurrentSkipListSet			
{add(0);contains(1)} || {add(1);**clear**();remove(1);contains(1)}	T,T,T,N,F,T	weak	398 / 2,589,570
{remove(0);remove(1)} || {add(0);add(1);**headSet**(2)}	T,T,T,T,[0]	peer	106 / 1,643,840
{add(0);add(1);contains(0)} || {add(1);**pollFirst**();remove(1)}	T,F,T,T,1,F	weak	22 / 1,955,120
{add(1);add(0);contains(0);add(0)} || {add(0);**pollLast**()}	T,F,F,T,T,0	weak	8 / 2,128,990
{add(0);remove(1)} || {add(1);**size**()}	T,T,T,0	peer	1 / 2,785,410
{**subSet**(0,3)} || {add(1);add(0);add(2)}	{1,2},T,T,T	peer	836 / 1,970,550
{**tailSet**(0)} || {add(1);add(0);remove(0);remove(1)}	{0},T,T,T,T	peer	631 / 2,340,020
{remove(0);remove(1)} || {add(0);add(1);**toArray**()}	T,T,T,T,[0]	peer	1 / 739,050
{add(0);add(1);**toString**()} || {remove(0);remove(1)}	T,T,[0],T,T	peer	23 / 1,843,310
ConcurrentLinkedQueue			
{peek();**clear**()} || { offer(0);peek();offer(0);poll()}	N,N,T,N,T,N	weak	8 / 2,646,350
{poll();offer(0)} || {offer(1);**size**()}	1,T,T,2	peer	3 / 3,458,050
{**toArray**()} || {offer(1);poll();offer(0)}	[1,0],T,1,T	peer	23 / 1,340,340
{offer(0);poll();offer(0)} || {**toString**()}	T,0,T,[0,0]	peer	21 / 3,845,190
LinkedTransferQueue			
{**clear**()} || {offer(0);poll();offer(1);peek();peek()}	N,T,N,T,1,N	weak	6 / 1,996,060
{**size**()} || {offer(0);poll();offer(1)}	2,T,0,T	peer	13 / 3,475,350
{**toArray**()} || {offer(1);poll();offer(0)}	[1,0],T,1,T	peer	103 / 322,400
{poll();offer(1)} || {offer(1);**toString**()}	1,T,T,[1,1]	peer	55 / 1,867,010
LinkedBlockingQueue			
{offer(0);**peek**();poll()} || {poll();poll();offer(1)}	T,1,N,0,N,T	weak	2 / 2,276,340
ConcurrentLinkedDeque			
{poll();peek();poll()} || {**addFirst**(1);peek();offer(0)}	0,1,1,N,1,T	weak	1 / 3,666,240
{offer(0);**clear**()} || {offer(0);peek();offer(1);poll()}	T,N,T,N,T,N	weak	2,555 / 1,994,800
{offer(1);**getLast**()} || {offer(0);poll()}	T,E,T,1	peer	18 / 4,236,420
{poll();poll()} || {**offerFirst**(0);peek();offer(1);offer(1)}	1,0,T,0,T,T	weak	4 / 2,181,330
{offer(1);**peekLast**()} || {offer(0);poll()}	T,N,T,1	peer	22 / 3,540,930
{offer(0); offer(1); peek(); poll()} || { **pollLast**()}	T,T,0,1,0	weak	644 / 2,921,465
{peek(); **removeLastO..**(1); poll()} || {offer(1); offer(1); poll()}	1,F,1,T,T,1	weak	1 / 3,933,030
{offer(0); poll() } || {offer(1); **removeLast**(); poll()}	T,1,T,E,0	weak	164 / 3,032,894
{offer(1);poll();offer(1)} || {**size**()}	T,1,T,2	peer	1 / 3,817,410
{peek(); offer(1); offer(0); **toArray**()} || {poll(); poll()}	N,T,T,[1],1,0	basic	68 / 2,156,940
{poll(); poll()} || {offer(1); offer(0);**toString**()}	1,0,T,T,[1]	basic	4 / 1,236,452

Table 3. Relaxed-visibility annotations for Java concurrent objects.

	ConcurrentHashMap
visibility	methods
complete	put, get, remove, containsKey, replace, putIfAbsent
monotonic	contains, containsValue, keys, values, elements, entrySet, keySet, toString
weak	isEmpty
none	clear, size, mappingCount
	ConcurrentSkipListMap
complete	put, get, remove, putIfAbsent, isEmpty, containsKey, replace, *floorKey*, *firstKey*, *ceilingKey*, *lastKey*, *higherKey*, *lowerKey*, *keySet*, *headMap*
monotonic	containsValue, entrySet, size, tailMap, toString, values
none	clear, pollFirstEntry, pollLastEntry
	ConcurrentSkipListSet
complete	add, remove, contains, isEmpty, *ceiling*, *floor*, *first*, *last*, *lower*, *higher*
monotonic	headSet, subSet, tailSet, size, toArray, toString
none	clear, pollFirst, pollLast

	ConcurrentLinkedQueue, LinkedTransferQueue
visibility	methods
complete	offer, peek, poll, add, isEmpty, remove, contains, remove(Object), element
monotonic	size, toArray, toString
none	clear
	LinkedBlockingQueue
complete	offer, poll, add, put, take, clear, isEmpty, remove, element, size, toArray, toString, remove(Object), contains, remainingCapacity
none	peek
	ConcurrentLinkedDeque
complete	offer, peek, poll, add, addLast, isEmpty, offerLast, getFirst, remove, removeFirst, remove(Object), peekFirst, element, pollFirst, contains, removeFirstOccurrence
monotonic	getLast, peekLast, size
weak	toArray, toString
none	clear, addFirst, offerFirst, pollLast, removeLast, removeLastOccurrence

given violation, the probability of this diminishes as we generate more and more programs within given bounds, i.e., on threads, invocations, and values. In our study, this second step was somewhat superfluous, since we did not encounter a case where some consistency violation manifests only with two non-atomic methods; every violation to a given method m's annotation also surfaced in programs where m was the only non-atomic method.

It is interesting to note that among the classes evaluated, their strongest visibility specifications use only the complete, monotonic, and weak annotations, leaving absent causal, peer, and basic. While we cannot soundly conclude their absence, e.g., across all Java classes, we conjecture that objects designed around conventional single- and multi-core microprocessor architectures may not generally exploit the optimizations which are characterized by the causal and peer visibilities. These visibilities are likely to be more prevalent in distributed and non-uniform memory access (NUMA) architectures, where replication and messaging is employed in place of shared-memory access. In these settings, the causal and peer visibilities naturally capture consistency mechanisms based on the transmission of messages. Relaxing monotonic to basic visibility captures, e.g., replica storage optimizations which disregard previously-received messages. In any case, strongest-possible specifications are often undesired since they constrain future optimizations. It is conceivable that architects might opt for, e.g., causal over complete visibility, to avoid imposing synchronization bottlenecks in the specification.

ConcurrentHashMap: contains

`{ put(0,0) } || { remove(1) } || { put(1,0); contains(0) }`

outcome	atomic?	frequency
`null, null, null, true`	✓	2,621,646
`null, 0, null, true`	✓	134,083
`null, 0, null, false`	✓	11

ConcurrentHashMap: contains

`{ put(0,0); remove(1) } || { put(1,0); contains(0) }`

outcome	atomic?	frequency
`null, null, null, true`	✓	1,224,150
`null, 0, null, true`	✓	1,827,063
`null, 0, null, false`	×	7

Fig. 10. While SHBC cannot identify atomicity violations in executions of the left program, in which the first thread's put is real-time ordered before the second thread's remove, and contains returns false, SHBC does identify the same violation in executions of the right program, which imposes the essential ordering constraint on program order.

5.3 Efficacy of Sequential Happens-Before Consistency

Our third finding is that SHBC-based validation is an effective means of exposing consistency violations. Our experience demonstrates that consistency violations are readily witnessed in small programs, quickly, without the additional resolution provided by real-time order.

As argued in Section 4, we can explain the robustness of SHBC as follows: even if SHBC cannot identify a consistency violation in the executions of a given program, i.e., since a crucial real-time ordering is unobserved, SHBC may identify the same violation in the executions of another program, i.e., whose happens-before, or even program order, witnesses the crucial ordering constraint. For instance, consider the SHBC-based test results of the two programs in Figure 10. The left program exhibits an atomicity violation in executions in which contains returns false, and the first thread's put operation is ordered in real-time before the second thread's remove operation. Without observing this real-time order, this outcome is consistent, since the first thread's put operation could have executed after contains, which may observe an empty map. However, SHBC can identify this very same atomicity violation in the right program, since the crucial ordering constraint between put and remove is enforced by the program order of the first thread. In fact, this is even a violation of sequential consistency. While the theoretical soundness of SHBC for determining linearizability is unclear, since it requires some mechanism by which programs can witness arbitrary real-time orders without perturbing, e.g., possible weak-memory effects — see Section 4 — our results and experience suggest that SHBC is effective at exposing violations in practice, without the need to monitor real-time invocation order.

Besides the ability to recognize violations, SHBC-based testing can be performed efficiently, with off-the-shelf concurrency testing frameworks. By pre-computing consistent outcomes/behaviors prior to testing a given program, SHBC enables the exploration of millions of executions per second, avoiding the imposition of overhead due to recording the real-time order among operations, and computing linearizations per execution. This distinction is significant, since prior linearization-based approaches spend over 30 seconds per program to uncover violations (see Section 7) instead of the 1 second our technique spends. Furthermore, our experience is that memory-based program instrumentation for recording the real-time order, required by linearizability checking, interferes with potential weak-memory reorderings, thus suppressing the expression of weak-memory behaviors. While the use of hardware timers in place of memory-based instrumentation could alleviate this interference in principle, we have found that among the available timers: millisecond timers are too imprecise to witness real-time orderings in the small programs we tested; high-precision timers are too costly, in terms of cycle counts, and thus suppress the expression of interesting interleavings between the methods under test. Note that while our simplified implementation checks only the outcomes of programs without additional happens-before constraints, the extension to behaviors of programs with additional memory operations is straightforward, and left for future work. In our experience such extensions do not incur significant overhead in our testing methodology.

5.4 Efficacy of Randomized Stress Testing as Validation

While we currently lack techniques capable of *proving* the validity of a given relaxed-visibility specification, our experience has instilled a certain amount of confidence based on two key factors. First, we observe that stress testing is effective exposing a wide range of thread interleavings and weak-memory effects, ultimately leading to the expression of possible consistency violations. Even one second of stress testing per program exercises millions of executions, which makes the expression of such violations fairly likely in practice.

The second major confidence-instilling factor is in our automated random enumeration of programs. Besides the need for automated enumeration, which we discuss next, we have found random generation particularly useful in discovering violations quickly, since systematic enumerations tend to get stuck for long periods of time exploring spaces of very similar programs which fail to express violations for similar reasons. Besides avoiding such locally-consistent program spaces, random enumeration is also less prone to the over-exploration of symmetric programs, e.g., differing only in the order of threads in program text, e.g., equivalent up to renaming of argument values.

More generally speaking, automated enumeration is essential since manually constructing programs which express inconsistent behaviors is infeasible without deeply understanding the implementation under test. The expression of consistency violations is fairly brittle, since very small program variations often suppress violations. For instance, Figure 11 lists 16 variations on the same program with different argument values, only 4 of which exhibit atomicity violations. Essentially, this program exhibits a bug in the clear method of Java 8's ConcurrentSkipListMap broke fundamental data-structure invariants when invoked concurrently with other methods, allowing subsequent insertions to disappear. The top-left program exhibits the corresponding consistency violation, by invoking containsKey on the key inserted after returning from clear. Besides the knowledge or luck required to consider this test program in the first place, additional knowledge or luck is required for picking the right argument values. To begin with, there are 16 other programs not considered, which do not exhibit violations, in which the keys passed to put and containsKey are distinct. Of the 16 remaining programs depicted, 8 of these (bottom of figure) pass the same key to the concurrent put operations, masking the bug. Only half of the remaining 8 programs expose the violation, due to subtleties around key ordering in the ConcurrentSkipListMap's underlying data structures: choosing key 1 exposes the violation, while key 0 does not. In our experience, this sort of implementation dependency makes the manual derivation of tests infeasible, yet makes automated random generation rather effective.

6 OPTIMALITY OF VISIBILITY SPECIFICATION

Given the simplicity of our visibility-annotation language, one may wonder whether this simplicity comes at the cost of expressive power: perhaps a more nuanced annotation language could provide stronger specifications which further boost clients' reasoning ability.

In this section we argue for a relative optimality of our annotation language, by considering alternative languages which are more expressive along a few different axes. We refer to the total order *lin* in a linearization $\ell = \langle lin, vis \rangle$ as the *linearization order*. In particular, we consider specification mechanisms which allow for the relaxation of linearization orders as well as visibilities (§6.1), joint constraints between linearization orders and visibilities (§6.2), and pairwise method annotations enabling target-dependent visibility (§6.3). We characterize these extensions in Section 6.4 with a fine-grained axiomatic specification language, which enables the systematic exploration of

ConcurrentSkipListMap: containsKey

`{ put(0,x) } || { clear(); put(1,y); containsKey(1) }`

outcome	atomic?	frequency
`null, _, null, true`	✓	99.4%
`null, _, null, false`	×	0.6%

`{ put(1,x) } || { clear(); put(0,y); containsKey(0) }`

outcome	atomic?	frequency
`null, _, null, true`	✓	100%

`{ put(1,x) } || { clear(); put(1,y); containsKey(1) }`

outcome	atomic?	frequency
`null, _, null, true`	✓	98.2%
`y, _, null, true`	✓	1.5%
`null, _, x, true`	✓	0.3%

`{ put(0,x) } || { clear(); put(0,y); containsKey(0) }`

outcome	atomic?	frequency
`null, _, null, true`	✓	99.8%
`y, _, null, true`	✓	0.2%
`null, _, x, true`	✓	3e-4%

Fig. 11. Only 4 of 16 argument-value variations of the depicted program, where $x, y \in \{0, 1\}$, exhibit atomicity violations; the other 16 argument-value variations (not depicted) exhibit no violations.

finer-grained specifications. Empirically, we find that despite the additional expressive power, the visibility specifications for Java concurrent objects derived in our simple annotation language, as reported in Section 5, correspond to maximally-strong specifications in this finer-grained language. While this does not preempt the general possibility of stronger consistency specifications, e.g., by considering ad-hoc annotation predicates rather than a fixed set of keywords, we believe this result is anyhow a meaningful indicator of optimality.

6.1 Linearization Relaxation

As a first language refinement, we consider allowing linearization orders which are not consistent with happens-before. While enabling great flexibility, we observe few cases in which the behaviors allowed by such relaxed linearization orders are not also allowed by other linearizations consistent with happens-before but with possibly more relaxed visibilities. For instance, consider again the { put(1,0); contains(0) } || { put(0,0); put(1,1) } program of Figure 2, whose outcome ⟨`null`, `false`, `null`, `0`⟩ is explained by the relaxed linearization order:

contains(0); put(1,0); put(0,0); put(1,1),

in which the first thread's invocations are reordered, together with a complete visibility where each invocation sees all its predecessors in the sequence. However, as argued in Section 2, this outcome is also explained by relaxing the visibility of contains in the linearization order:

put(1,0); put(0,0); put(1,1); contains(0)

such that it does not see put(0,0).

Allowing behaviors which are only justified by linearization relaxations is not ideal because it complicates client reasoning. For instance, consider this ConcurrentLinkedDeque client:[26]

{ offer(0); poll() } || { offer(1); removeLast(); poll() }

The outcome ⟨`true`, `1`, `true`, `NoSuchElementException`[27], `0`⟩ is only explained by the linearization:

offer(1); offer(0); poll(); poll(); removeLast(),

[26]The ConcurrentLinkedDeque is a double-ended queue with offer and poll methods for adding and removing elements, respectively, to the back and from the front of the queue. The removeLast method removes elements from the back of the queue.

[27]`NoSuchElementException` indicates an empty queue.

where the removeLast and poll invocations of the second thread are reordered (each invocation sees all its predecessors in the sequence).[28] For such clients, the main difficulty is designing reasoning principles that don't rely on the syntactic structure of the program, e.g., based on validating standard assertions in the client code, which is similar in spirit to the issues addressed in the context of reasoning about programs running on weak-memory platforms [Alglave and Cousot 2017; Lahav and Vafeiadis 2015; Turon et al. 2014].

6.2 Relating Linearizations And Visibilities

As a second refinement, we consider additional predicates relating visibilities and linearization orders. First, discerning whether the visibility of an invocation is a prefix of the linearization order, rather than a subsequence, as in the definition of the visibility mapping. Second, discerning whether the visibility of invocations increases according to their order in the linearization. These relations are formalized as the following visibility predicates:

$$\mathsf{linPrefix}(i, \ell) \Leftrightarrow \forall j \in \mathit{vis}(i).\ \mathit{lin}(j) \leq \mathit{vis}(i)$$
$$\mathsf{linMonotonic}(i, \ell) \Leftrightarrow \forall j \in \mathit{lin}(i).\ (\mathit{vis}(j) \setminus \{j\}) \leq \mathit{vis}(i)$$

where i is an invocation and $\ell = \langle \mathit{lin}, \mathit{vis} \rangle$ is a linearization. For instance, consider the non-relaxed linearization order of the contains method from Section 6.1, and the visibility mapping where contains does not see put(0,0). The invocation of contains does not satisfy linPrefix since its visibility is not a *complete* prefix of the linearization order (it sees put(1,1) but not its predecessor put(0,0) in the linearization order), and also, it does not satisfy linMonotonic since the visibility of put(1,1) is complete, and thus includes put(0,0), but the visibility of its successor contains(0) in the linearization order does not contain put(0,0). The predicate linPrefix corresponds to *prefix consistency* [Terry et al. 1995], a consistency criterion used in the context of distributed databases, while linMonotonic is analogous to monotonic, except that the visibility "grows" with respect to the linearization order instead of the happens-before order.

These predicates may offer finer-grain guarantees about the consistency of the API, but they are more difficult to reason about. The predicates we chose to include in our visibility specifications constrain the visibility in terms of the happens-before order, which we assume that it can be derived syntactically, rather than the linearization order which is related to the semantics of the implementation and changes from behavior to behavior. Furthermore, since the visibility mapping is consistent with the linearization order, i.e., $\mathit{vis}(i)$ is a subsequence of $\mathit{lin}(i)$, and the linearization order is consistent with happens-before, these relations between visibility and linearization can be inferred from the visibility predicates defined in Section 3 in all cases except for causal (note however that causal is not exercised by the Java objects we considered). For instance, let i be an invocation that satisfies $\mathsf{weak}(i, \ell, b)$, for some linearization ℓ of a behavior b, and does not satisfy all the predicates stronger than and including $\mathsf{basic}(i, \ell, b)$. Since $\mathsf{linPrefix}(i, \ell)$ implies $\mathsf{causal}(i, \ell, b)$ for any behavior b (because $\mathit{vis}(j)$ is a subsequence of $\mathit{lin}(j)$), we get that i does not satisfy $\mathsf{linPrefix}(i, \ell)$. Also, $\mathsf{linMonotonic}(i, \ell)$ implies a weakening of $\mathsf{monotonic}(i, \ell, b)$ defined by

$$\mathsf{monotonic}^*(i, \ell, b) \Leftrightarrow \forall j \in \mathit{hb}(i).\ (\mathit{vis}(j) \setminus \{j\}) \leq \mathit{vis}(i)$$

which excludes the guarantees of basic, i.e., we have that $\mathsf{monotonic}(i, \ell, b) \Leftrightarrow (\mathsf{monotonic}^*(i, \ell, b) \wedge \mathsf{basic}(i, \ell, b))$ for every b. This entailment holds because $\mathit{hb}(i)$ is a subsequence of $\mathit{lin}(i)$. Therefore, i does not satisfy $\mathsf{linMonotonic}(i, \ell)$. A similar reasoning can be applied when the strongest predicate satisfied by i is basic, monotonic, peer, or complete.

[28] We assume that the invocations of offer and poll, which are intended to be atomic, have a complete visibility.

6.3 From Unary to Binary Visibility Predicates

The visibility predicates define lower bounds for invocations' visibilities. For instance, basic(i, ℓ, b) requires invocation i to see at least its happens-before predecessors. These lower bounds could be weakened by keeping only invocations of certain methods, e.g., redefining basic(i, ℓ, b) as basic(i, ℓ, b, m) to say that i sees at least all the invocations of m which precede it in happens-before. For instance, consider the following program of ConcurrentHashMap:

{ get(0); put(1,0); isEmpty() } || { get(1); put(1,1); elements()}.

The observed outcome ⟨`null`, `1`, `true`, `null`, `null`, `[1]`⟩ could be explained by the linearization

get(0); get(1); put(1,1); elements(); put(1,0); isEmpty()

where all invocations besides isEmpty have complete visibility; isEmpty sees all invocations except for put(1,1) and put(1,0), the latter being a predecessor in program order (and happens-before). This outcome shows that isEmpty does not satisfy basic. However, letting i denote the invocation of isEmpty in this linearization, the predicate basic(i, ℓ, b, get) defined above would hold since i sees the invocation of get which precedes it in program order (and happens-before). The same transformation from unary to binary predicates can be also applied to the other predicates besides basic. Such binary predicates enable more precise specifications at the expense of rendering these specifications much more difficult to apprehend, e.g., disabling a modular documentation of the API, where each method is specified in isolation.

6.4 Systematic Exploration of Fine-Grained Specifications

To reason about the possible (combinations of) refinements to our visibility specifications, we consider a language of consistency models defined by axioms which characterize linearizations of behaviors. For simplicity, these axioms use a representation of the visibility mapping *vis* as a binary relation between invocations defined by $\{\langle i, j\rangle : i \in vis(j) \text{ and } i \neq j\}$. By an abuse of notation, this binary relation is named *vis* as well. The *(left) composition* $R_1 \circ R_2$ of two binary relations R_1 and R_2 is the set of pairs $\langle x, z\rangle$ such that $\langle x, y\rangle \in R_1$ and $\langle y, z\rangle \in R_2$ for some y. We denote the identity binary relation $\{\langle x, x\rangle : x \in X\}$ on a set X by $[X]$, and we write $[x]$ to denote $[\{x\}]$.

The following grammar of *consistency axioms* ϕ is parameterized by *sets of method names* M characterizing sets of invocations in a given abstract execution:

$$\phi ::= qrel \supseteq rel \qquad qrel ::= \mathsf{lin} \mid \mathsf{vis} \qquad rel ::= qrel \mid \mathsf{hb} \mid [M] \mid rel \circ rel$$

M is interpreted as the set of M-invocations. A *consistency model* Φ is a set $\{\phi_1, \phi_2, \ldots\}$ of consistency axioms. The interpretation of these axioms over pairs of behaviors and linearizations of a program $p = \langle po, hbs\rangle$ is defined as expected:

$$\begin{aligned}&\langle\langle hb, ret\rangle, \langle lin, vis\rangle\rangle \models \phi(\vec{M})\\ &\text{ iff } \phi[hb/\mathsf{hb}][lin/\mathsf{lin}][vis/\mathsf{vis}][\{i \in po : i \text{ is an } M_j\text{-invocation }\}/M_j]_j \text{ is valid}\end{aligned}$$

We extend this semantics to consistency models as $\langle b, \ell\rangle \models \Phi$ iff $\langle b, \ell\rangle \models \phi$ for all $\phi \in \Phi$. We say that consistency model Φ_1 is *stronger* than Φ_2, written $\Phi_1 \Rightarrow \Phi_2$, when $\langle b, \ell\rangle \models \Phi_1$ implies $\langle b, \ell\rangle \models \Phi_2$ for every behavior-linearization pair $\langle b, \ell\rangle$. For a consistency model Φ, we say that a behavior b of a program p over abstract data type A is *expected* when there exists a linearization ℓ of b admitted by A such that $\langle b, \ell\rangle$ satisfies Φ. An implementation *Impl* of A *satisfies* Φ when each observed behavior is expected, for every program over A.

We have considered a bounded space of consistency models, each containing the axiom $\mathsf{lin} \supseteq \mathsf{vis}$. The remaining axioms are instantiated from the schemas enumerated in Table 4. These axioms describe both the visibility specifications of Section 3, as well as the refinements outlined in

Table 4. Consistency axiom schemas.

schema name	definition
HbConsistentLin(M_1, M_2)	lin $\supseteq [M_1] \circ$ hb $\circ [M_2]$
ReadPreviousOps(M_1, M_2)	vis $\supseteq [M_1] \circ$ hb $\circ [M_2]$
HbMonotonicReads(M_1, M_2)	vis $\supseteq [M_1] \circ$ vis $\circ$ hb $\circ [M_2]$
ReadHbPrefix(M_1, M_2)	vis $\supseteq [M_1] \circ$ hb $\circ$ vis $\circ [M_2]$
SingleOrder(M_1, M_2)	vis $\supseteq [M_1] \circ$ lin $\circ [M_2]$
LinMonotonicReads(M_1, M_2)	vis $\supseteq [M_1] \circ$ vis $\circ$ lin $\circ [M_2]$
ReadLinPrefix(M_1, M_2)	vis $\supseteq [M_1] \circ$ lin $\circ$ vis $\circ [M_2]$
TransVis(M_1, M_2)	vis $\supseteq [M_1] \circ$ vis $\circ$ vis $\circ [M_2]$
M_1, M_2 are sets of method names	

Table 5. A strongest consistency model for Java concurrent collections, where M_w, M_m, and M_c are the sets of weak, monotonic, and respectively, complete methods defined in Table 3.

{ lin $\supseteq$ vis,
HbConsistentLin($M_w \cup M_m \cup M_c$, $M_w \cup M_m \cup M_c$),
ReadPreviousOps($M_w \cup M_m \cup M_c$, $M_m \cup M_c$),
HbMonotonicReads($M_w \cup M_m \cup M_c$, $M_m \cup M_c$),
ReadHbPrefix($M_w \cup M_m \cup M_c$, M_c),
TransVis($M_w \cup M_m \cup M_c$, M_c),
SingleOrder($M_w \cup M_m \cup M_c$, M_c),
LinMonotonicReads($M_w \cup M_m \cup M_c$, M_c),
ReadLinPrefix($M_w \cup M_m \cup M_c$, M_c)}

Sections 6.1–6.3. Thus, a visibility specification S over methods M can be expressed as a consistency model Φ which includes lin $\supseteq$ vis, HbConsistentLin(M, M), as well as:

$$\begin{array}{l} \text{ReadPreviousOps}(M, m), \text{ for each } m \text{ such that } S(m) \in \{\text{basic, monotonic, peer, causal}\} \\ \text{HbMonotonicReads}(M, m), \text{ for each } m \text{ such that } S(m) \in \{\text{monotonic, peer}\} \\ \text{ReadHbPrefix}(M, m), \text{ for each } m \text{ such that } S(m) \in \{\text{peer}\} \\ \text{TransVis}(M, m), \text{ for each } m \text{ such that } S(m) \in \{\text{causal}\} \\ \text{SingleOrder}(M, m), \text{ for each } m \text{ such that } S(m) \in \{\text{complete}\} \end{array}$$

To compare the expressive power of this refined language of consistency models with our simple annotation language, we have conducted a systematic exploration of the maximally-strong consistency models. We have automated this exploration by implementing a testing-based procedure, similarly to the validator described in Section 5, to derive maximal models by filtering out consistency axioms which are inconsistent with observed behaviors. Generally speaking, an API implementation may satisfy multiple incomparable consistency models, and we have implemented certain heuristics to guide the selection among maximals according to the discussions of Sections 6.1–6.3, e.g., we prefer avoiding linearization order relaxations. According to these heuristics, we have automatically derived one particular maximal consistency model, among other maximals, for each of the Java classes evaluated in Section 5, which is no stronger than the specification derived from our simple annotation language, as listed in Table 3. For each class, this derived consistency model corresponds to an instantiation of the schema of Table 5: the sets M_w, M_m, and M_c are precisely the sets of methods with weak, monotonic, and complete visibility annotations, respectively, for a given class, as listed in Table 3. This correspondence demonstrates an empirical optimality of our simple annotation language: even this expressive language of consistency models cannot derive stronger specifications for the existing Java concurrent objects.

7 RELATED WORK

Our specification methodology follows a long line of work around consistency criteria for concurrent objects. Herlihy and Wing [1990] described linearizability, the de-facto criterion. While Batty et al. [2013] adapt linearizability to a single modern platform, C/C++, we describe a general adaptation by abstracting from the platform-specifics of the happens-before (HB) order among API invocations. Motivated by replication-based distributed systems, Burckhardt [2014]; Burckhardt et al. [2014] describe a more general axiomatic framework for specifying weaker consistencies like eventual consistency [Terry et al. 1995] and causal consistency [Ahamad et al. 1995; Lamport 1978]. While these axiomatic specifications facilitate general reasoning for uniformly-weak operations, they do not enable the specification of objects with operations of varying consistencies. Sergey et al. [2016] describe API-specific per-operation consistency specifications which are not portable across APIs. In contrast, the annotation keywords of our relaxed-visibility specifications are *generic*: their meaning is independent from the functional specifications of the APIs they annotate.

Recently Doherty et al. [2018]; Dongol et al. [2018] proposed a notion of *causal happens-before linearizability* which appears to be equivalent to *complete* sequential happens-before consistency (SHBC), i.e., without visibility relaxation, for APIs which do not contribute to happens-before. Their definition also requires implementations to guarantee specified happens-before contributions — e.g., that the insertion of a given element in a Java collection happens-before its corresponding retrieval — which is orthogonal to return-value consistency. Beyond the similarity of these definitions, our scientific contributions are distinct. On the one hand, Dongol et al. have demonstrated how an implementation may guarantee its specified happens-before contributions, and that their definition admits compositionality for APIs meeting a certain condition: non-commutative operations are related by happens-before. On the other hand, we demonstrate that SHBC is an effective criterion for real platforms like Java, in that it admits sound and efficient per-execution validation procedures (unlike linearizability — see §4), and actually holds (modulo visibility relaxation) over all observed executions of several high-performance implementations. On the contrary, it is not apparent that Dongol et al.'s compositionality condition is met by popular API specifications: Java collections only order operations accessing the same element, yet many multi-element operations are non-commutative, e.g., queue insertions, e.g., size methods.

Our concurrent-object testing methodology joins many others. Pradel and Gross [2012] consider *thread safety*, which prohibits unexpected exceptions and deadlocks, but not inconsistency with functional specifications. Samak and Ramanathan [2015] consider the atomicity of lock-based implementations, ignoring, e.g., the majority of Java's concurrent objects. Shacham et al. [2011] exploit commutativity for testing linearizability of operations composed of other linearizable objects. Line-Up [Burckhardt et al. 2010] checks linearizability by enumerating linearizations per execution, spending an average of 31.5s per test program, a cost we avoid by precomputing expected behaviors; spending just 1s per program effectively speeds-up Line-Up by 30×. Moreover, Line-Up's model checker can interfere with weak-memory behaviors, and test generation requires user-specified argument-value ranges. We avoid invasive recording of real-time order, and include argument values in the test discovery process, exposing violations which only occur with unexpected combinations.

Our testing-based methodology is perhaps most similar to litmus testing for validating hardware memory-consistency specifications, which checks the observed outcomes from a given program against a precomputed set of expected outcomes, and can be applied to the validation of sequential consistency [Alglave et al. 2014; Mador-Haim et al. 2010]. Our approach essentially generalizes litmus testing from memory operations to the operations of arbitrary ADTs, and from return-value outcomes to behaviors, i.e., including an underlying platform-defined happens-before ordering.

8 CONCLUSION

Our specification methodology enables the precise specification of software modules whose operations provide varying consistency guarantees. Besides delineating boundaries against which to validate potential optimizations, our methodology enables module clients to establish program properties which rely on precise module guarantees. Without such a characterization of relaxed consistency, the invocation of even a single non-atomic operation essentially forfeits any guarantees for all operations. In contrast, our simple annotation language yields precise yet generic consistency guarantees for actual software APIs with operations of varying consistencies. To the best of our knowledge, we are the first to develop such a methodology.

While Java's concurrency API already justifies our simplified categorization of relaxed consistencies, our experience suggests that further refinement is called for. In particular, we expect that some of the methods which fall below our baseline of weak visibility (see §5) are correct, in the sense that they actually exhibit their intended consistencies. We believe that extending our approach to handle *bulk mutators* like the clear method, which modify multiple collection elements, could assign meaningful consistencies, avoiding their total loss of guarantees. It is important to distinguish these cases from other losses of consistency, which indicate implementation bugs.

ACKNOWLEDGMENTS

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

REFERENCES

Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 3–18. http://dl.acm.org/citation.cfm?id=3009883

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

Mark Batty, Mike Dodds, and Alexey Gotsman. 2013. Library abstraction for C/C++ concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 235–248. https://doi.org/10.1145/2429069.2429099

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (2014), 1–150. https://doi.org/10.1561/2500000011

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 330–340. https://doi.org/10.1145/1806596.1806634

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 271–284. https://doi.org/10.1145/2535838.2535848

Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2018. Making Linearizability Compositional for Partially Ordered Executions. In *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings (Lecture Notes in Computer Science)*, Carlo A. Furia and Kirsten Winter (Eds.), Vol. 11023. Springer, 110–129. https://doi.org/10.1007/978-3-319-98938-9_7

Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science)*, Isil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer, 183–204. https://doi.org/10.1007/978-3-319-73721-8_9

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.), Vol. 9135. Springer, 311–323. https://doi.org/10.1007/978-3-662-47666-6_25

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 378–391. https://doi.org/10.1145/1040305.1040336

Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. 2016. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 26:1–26:12. https://doi.org/10.1145/2851141.2851170

Michael Pradel and Thomas R. Gross. 2012. Fully automatic and precise detection of thread safety violations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 521–530. https://doi.org/10.1145/2254064.2254126

Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 131–142. https://doi.org/10.1145/2786805.2786874

Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. 2016. Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 92–110. https://doi.org/10.1145/2983990.2983999

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 51–64. https://doi.org/10.1145/2048066.2048073

Inc SPARC International. 1994. The SPARC Architecture Manual Version 9. (1994).

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994.* IEEE Computer Society, 140–149. https://doi.org/10.1109/PDIS.1994.331722

Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, Michael B. Jones (Ed.). ACM, 172–183. https://doi.org/10.1145/224056.224070

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 691–707. https://doi.org/10.1145/2660193.2660243