Verifying Visibility-Based Weak Consistency

Siddharth Krishna[1], Michael Emmi[2], Constantin Enea[3], and Dejan Jovanović[2]

[1] New York University, New York, NY, USA, siddharth@cs.nyu.edu
[2] SRI International, New York, NY, USA, michael.emmi@gmail.com, dejan.jovanovic@sri.com
[3] Université de Paris, IRIF, CNRS, F-75013 Paris, France, cenea@irif.fr

Abstract. Multithreaded programs generally leverage efficient and thread-safe *concurrent objects* like sets, key-value maps, and queues. While some concurrent-object operations are designed to behave atomically, each witnessing the atomic effects of predecessors in a linearization order, others forego such strong consistency to avoid complex control and synchronization bottlenecks. For example, contains (value) methods of key-value maps may iterate through key-value entries without blocking concurrent updates, to avoid unwanted performance bottlenecks, and consequently overlook the effects of some linearization-order predecessors. While such *weakly-consistent* operations may not be atomic, they still offer guarantees, e.g., only observing values that have been present.

In this work we develop a methodology for proving that concurrent object implementations adhere to weak-consistency specifications. In particular, we consider (forward) simulation-based proofs of implementations against *relaxed-visibility specifications*, which allow designated operations to overlook some of their linearization-order predecessors, i.e., behaving as if they never occurred. Besides annotating implementation code to identify *linearization points*, i.e., points at which operations' logical effects occur, we also annotate code to identify *visible operations*, i.e., operations whose effects are observed; in practice this annotation can be done automatically by tracking the writers to each accessed memory location. We formalize our methodology over a general notion of transition systems, agnostic to any particular programming language or memory model, and demonstrate its application, using automated theorem provers, by verifying models of Java concurrent object implementations.

1 Introduction

Programming efficient multithreaded programs generally involves carefully organizing shared memory accesses to facilitate inter-thread communication while avoiding synchronization bottlenecks. Modern software platforms like Java include reusable abstractions which encapsulate low-level shared memory accesses and synchronization into familiar high-level abstract data types (ADTs). These so-called *concurrent objects* typically include mutual-exclusion primitives like locks, numeric data types like atomic integers, as well as collections like sets, key-value maps, and queues; Java's standard-edition platform contains many implementations of each. Such objects typically provide strong consistency guarantees like *linearizability* [18], ensuring that each operation appears to happen atomically, witnessing the atomic effects of predecessors according to some linearization order among concurrently-executing operations.

P. Müller (Ed.): ESOP 2020, LNCS 12075, pp. 280–307, 2020.
https://doi.org/10.1007/978-3-030-44914-8_11

While such strong consistency guarantees are ideal for logical reasoning about programs which use concurrent objects, these guarantees are too strong for many operations, since they preclude simple and/or efficient implementation — over half of Java's concurrent collection methods forego atomicity for *weak-consistency* [13]. On the one hand, basic operations like the get and put methods of key-value maps typically admit relatively-simple atomic implementations, since their behaviors essentially depend upon individual memory cells, e.g., where the relevant key-value mapping is stored. On the other hand, making aggregate operations like size and contains (value) atomic would impose synchronization bottlenecks, or otherwise-complex control structures, since their atomic behavior depends simultaneously upon the values stored across many memory cells. Interestingly, such implementations are not linearizable even when their underlying memory operations are sequentially consistent, e.g., as is the case with Java 8's concurrent collections, whose memory accesses are data-race free.[4]

For instance, the contains (value) method of Java's concurrent hash map iterates through key-value entries without blocking concurrent updates in order to avoid unreasonable performance bottlenecks. Consequently, in a given execution, a contains-value-v operation o_1 will overlook operation o_2's concurrent insertion of $k_1 \mapsto v$ for a key k_1 it has already traversed. This oversight makes it possible for o_1 to conclude that value v is not present, and can only be explained by o_1 being linearized before o_2. In the case that operation o_3 removes $k_2 \mapsto v$ concurrently before o_1 reaches key k_2, but only after o_2 completes, then atomicity is violated since in every possible linearization, either mapping $k_2 \mapsto v$ or $k_1 \mapsto v$ is always present. Nevertheless, such weakly-consistent operations still offer guarantees, e.g., that values never present are never observed, and initially-present values not removed are observed.

In this work we develop a methodology for proving that concurrent-object implementations adhere to the guarantees prescribed by their weak-consistency specifications. The key salient aspects of our approach are the lifting of existing sequential ADT specifications via *visibility relaxation* [13], and the harnessing of simple and mechanizable reasoning based on *forward simulation* [25] by relaxed-visibility ADTs. Effectively, our methodology extends the predominant forward-simulation based linearizability-proof methodology to concurrent objects with weakly-consistent operations, and enables automation for proving weak-consistency guarantees.

To enable the harnessing of existing sequential ADT specifications, we adopt the recent methodology of *visibility relaxation* [13]. As in linearizability [18], the return value of each operation is dictated by the atomic effects of its predecessors in some (i.e., existentially quantified) linearization order. To allow consistency weakening, operations are allowed, to a certain extent, to overlook some of their linearization-order predecessors, behaving as if they had not occurred. Intuitively, this (also existentially quantified) *visibility* captures the inability or unwillingness to atomically observe the values stored across many memory cells. To provide guarantees, the extent of

[4] Java 8 implementations guarantee data-race freedom by accessing individual shared-memory cells with atomic operations via volatile variables and compare-and-swap instructions. Starting with Java 9, the implementations of the concurrent collections use the `VarHandle` mechanism to specify shared variable access modes. Java's official language and API specifications do not clarify whether these relaxations introduce data races.

visibility relaxation is bounded to varying degrees. Notably, the visibility of an *absolute* operation must include all of its linearization-order predecessors, while the visibility of a *monotonic* operation must include all happens-before predecessors, along with all operations visible to them. The majority of Java's concurrent collection methods are absolute or monotonic [13]. For instance, in the contains-value example described above, by considering that operation o_2 is not visible to o_1, the conclusion that v is not present can be justified by the linearization $o_2; o_3; o_1$, in which o_1 sees o_3's removal of $k_2 \mapsto v$ yet not o_2's insertion of $k_1 \mapsto v$. Ascribing the monotonic visibility to the contains-value method amounts to a guarantee that initially-present values are observed unless removed (i.e., concurrently).

While relaxed-visibility specifications provide a means to describing the guarantees provided by weakly-consistent concurrent-object operations, systematically establishing implementations' adherence requires a strategy for demonstrating *simulation* [25], i.e., that each step of the implementation is simulated by some step of (an operational representation of) the specification. The crux of our contribution is thus threefold: first, to identify the relevant specification-level actions with which to relate implementation-level transitions; second, to identify implementation-level annotations relating transitions to specification-level actions; and third, to develop strategies for devising such annotations systematically. For instance, the existing methodology based on *linearization points* [18] essentially amounts to annotating implementation-level transitions with the points at which its specification-level action, i.e., its atomic effect, occurs. Relaxed-visibility specifications require not only a witness for the existentially-quantified linearization order, but also an existentially-quantified visibility relation, and thus requires a second kind of annotation to resolve operations' visibilities. We propose a notion of *visibility actions* which enable operations to declare their visibility of others, e.g., specifying the writers of memory cells it has read.

The remainder of our approach amounts to devising a systematic means for constructing simulation proofs to enable automated verification. Essentially, we identify a strategy for systematically annotating implementations with visibility actions, given linearization-point annotations and visibility bounds (i.e., absolute or monotonic), and then encode the corresponding simulation check using an off-the-shelf verification tool. For the latter, we leverage CIVL [16], a language and verifier for Owicki-Gries style modular proofs of concurrent programs with arbitrarily-many threads. In principle, since our approach reduces simulation to safety verification, any safety verifier could be used, though CIVL facilitates reasoning for multithreaded programs by capturing interference at arbitrary program points. Using CIVL, we have verified monotonicity of the contains-value and size methods of Java's concurrent hash-map and concurrent linked-queue, respectively — and absolute consistency of add and remove operations. Although our models are written in CIVL and assume sequentially-consistent memory accesses, they capture the difficult aspects of weak-consistency in Java, including heap-based memory access; furthermore, our models are also sound with respect to Java 8's memory model, since their Java 8 implementations guarantee data-race freedom.

In summary, we present the first methodology for verifying weakly-consistent operations using sequential specifications and forward simulation. Contributions include:

- the formalization of our methodology over a general notion of transition systems, agnostic to any particular programming language or memory model (§3);
- the application of our methodology to verifying a weakly-consistent contains-value method of a key-value map (§4); and
- a mechanization of our methodology used for verifying models of weakly-consistent Java methods using automated theorem provers (§5).

Aside from the outline above, this article summarizes an existing weak-consistency specification methodology via visibility relaxation (§2), summarizes related work (§6), and concludes (§7). Proofs of all theorems and lemmas are listed in Appendix A.

2 Weak Consistency

Our methodology for verifying weakly-consistent concurrent objects relies both on the precise characterization of weak consistency specifications, as well as a proof technique for establishing adherence to specifications. In this section we recall and outline a characterization called *visibility relaxation* [13], an extension of sequential abstract data type (ADT) specifications in which the return values of some operations may not reflect the effects of previously-effectuated operations.

Notationally, in the remainder of this article, ε denotes the empty sequence, $\emptyset$ denotes the empty set, _ denotes an unused binding, and $\top$ and $\bot$ denote the Boolean values true and false, respectively. We write $R(x)$ to denote the inclusion $x \in R$ of a tuple x in the relation R; and $R[x \mapsto y]$ to denote the extension $R \cup \{xy\}$ of R to include xy; and $R \mid X$ to denote the projection $R \cap X^*$ of R to set X; and $\overline{R}$ to denote the complement $\{x : x \notin R\}$ of R; and $R(x)$ to denote the image $\{y : xy \in R\}$ of R on x; and $R^{-1}(y)$ to denote the pre-image $\{x : xy \in R\}$ of R on y; whether $R(x)$ refers to inclusion or an image will be clear from its context. Finally, we write x_i to refer to the ith element of tuple $x = x_0x_1 \ldots$.

2.1 Weak-Visibility Specifications

For a general notion of ADT specifications, we consider fixed sets $\mathbb{M}$ and $\mathbb{X}$ of method names and argument or return values, respectively. An *operation label* $\lambda = \langle m, x, y\rangle$ is a method name $m \in \mathbb{M}$ along with argument and return values $x, y \in \mathbb{X}$. A *read-only predicate* is a unary relation $R(\lambda)$ on operation labels, an *operation sequence* $s = \lambda_0\lambda_1 \ldots$ is a sequence of operation labels, and a *sequential specification* $S = \{s_0, s_1, \ldots\}$ is a set of operation sequences. We say that R is *compatible* with S when S is closed under deletion of read-only operations, i.e., $\lambda_0 \ldots \lambda_{j-1}\lambda_{j+1} \ldots \lambda_i \in S$ when $\lambda_0 \ldots \lambda_i \in S$ and $R(\lambda_j)$.

Example 1. The *key-value map* ADT sequential specification S_{m} is the prefix-closed set containing all sequences $\lambda_0 \ldots \lambda_i$ such that λ_i is either:

- $\langle \mathrm{put}, kv, b\rangle$, and $b = \top$ iff some $\langle \mathrm{rem}, k, \_\rangle$ follows any prior $\langle \mathrm{put}, kv, \_\rangle$;
- $\langle \mathrm{rem}, k, b\rangle$, and $b = \top$ iff no other $\langle \mathrm{rem}, k, \_\rangle$ follows some prior $\langle \mathrm{put}, kv, \_\rangle$;
- $\langle \mathrm{get}, k, v\rangle$, and no $\langle \mathrm{put}, kv', \_\rangle$ nor $\langle \mathrm{rem}, k, \_\rangle$ follows some prior $\langle \mathrm{put}, kv, \_\rangle$, and $v = \bot$ if no such $\langle \mathrm{put}, kv, \_\rangle$ exists; or

– $\langle \text{has}, v, b\rangle$, and $b = \top$ iff no prior $\langle \text{put}, kv', \_\rangle$ nor $\langle \text{rem}, k, \_\rangle$ follows some prior $\langle \text{put}, kv, \_\rangle$.

The read-only predicate R_m holds for the following cases:

$$R_\text{m}(\langle \text{put}, \_, b\rangle) \text{ if } \neg b \quad R_\text{m}(\langle \text{rem}, \_, b\rangle) \text{ if } \neg b \quad R_\text{m}(\langle \text{get}, \_, \_\rangle) \quad R_\text{m}(\langle \text{has}, \_, \_\rangle).$$

This is a simplification of Java's Map ADT, i.e., with fewer methods.[5]

To derive weak specifications from sequential ones, we consider a set $\mathbb{V}$ of exactly two *visibility* labels from prior work [13]: *absolute* and *monotonic*.[6] A *visibility annotation* $V : \mathbb{M} \to \mathbb{V}$ maps each method $m \in \mathbb{M}$ to a visibility $V(m) \in \mathbb{V}$.

Intuitively, absolute visibility requires operations to observe the effects of all of their linearization-order predecessors. The weaker monotonic visibility requires operations to observe the effects of all their happens-before (i.e., program- and synchronization-order) predecessors, along with the effects already observed by those predecessors, i.e., so that sets of visible effects are monotonically increasing over happens-before chains of operations; conversely, operations may ignore effects which have been ignored by their happens-before predecessors, so long as those effects are not transitively related by program and synchronization order.

Definition 1. *A* weak-visibility specification $W = \langle S, R, V\rangle$ *is a sequential specification* S *with a compatible read-only predicate* R *and a visibility annotation* V*.*

Example 2. The *weakly-consistent contains-value map* $W_\text{m} = \langle S_\text{m}, R_\text{m}, V_\text{m}\rangle$ annotates the key-value map ADT methods of S_m from Example 1 with:

$$V_\text{m}(\text{put}) = V_\text{m}(\text{rem}) = V_\text{m}(\text{get}) = \text{absolute}, \qquad V_\text{m}(\text{has}) = \text{monotonic}.$$

Java's concurrent hash map appears to be consistent with this specification [13].

We ascribe semantics to specifications by characterizing the values returned by concurrent method invocations, given constraints on invocation order. In practice, the *happens-before* order among invocations is determined by a *program order*, i.e., among invocations of the same thread, and a *synchronization order*, i.e., among invocations of distinct threads accessing the same atomic objects, e.g., locks. A *history* $h = \langle O, inv, ret, hb\rangle$ is a set $O \subseteq \mathbb{N}$ of numeric operation identifiers, along with an invocation function $inv : O \to \mathbb{M} \times \mathbb{X}$ mapping operation identifiers to method names and argument values, a partial return function $ret : O \rightharpoonup \mathbb{X}$ mapping operation identifiers to return values, and a (strict) partial happens-before relation $hb \subseteq O \times O$; the *empty history* $h_\emptyset$ has $O = inv = ret = hb = \emptyset$. An operation $o \in O$ is *complete* when $ret(o)$ is defined, and is otherwise *incomplete*; then h is *complete* when each operation is. The *label* of a complete operation o with $inv(o) = \langle m, x\rangle$ and $ret(o) = y$ is $\langle m, x, y\rangle$.

To relate operations' return values in a given history back to sequential specifications, we consider certain sequencings of those operations. A *linearization* of a history $h = \langle O, \_, \_, hb\rangle$ is a total order $lin \supseteq hb$ over O which includes hb, and a *visibility*

[5] For brevity, we abbreviate Java's remove and contains-value methods by rem and has.

[6] Previous work refers to absolute visibility as *complete*, and includes additional visibility labels.

projection vis of lin maps each operation $o \in O$ to a subset $vis(o) \subseteq lin^{-1}(o)$ of the operations preceding o in lin; note that $\langle o_1, o_2\rangle \in vis$ means o_1 observes o_2. For a given read-only predicate R, we say o's visibility is *monotonic* when it includes every happens-before predecessor, and operation visible to a happens-before predecessor, which is not read-only,[7] i.e., $vis(o) \supseteq \left(hb^{-1}(o) \cup vis(hb^{-1}(o))\right) \mid \overline{R}$. We says o's visibility is *absolute* when $vis(o) = lin^{-1}(o)$, and vis is itself *absolute* when each $vis(o)$ is. An *abstract execution* $e = \langle h, lin, vis\rangle$ is a history h along with a linearization of h, and a visibility projection vis of lin. An abstract execution is *sequential* when hb is total, *complete* when h is, and *absolute* when vis is.

Example 3. An abstract execution can be defined using the linearization[8]

$$\langle \text{put}, \langle 1, 1\rangle, \top\rangle\ \langle \text{get}, 1, 1\rangle\ \langle \text{put}, \langle 0, 1\rangle, \top\rangle\ \langle \text{put}, \langle 1, 0\rangle, \bot\rangle\ \langle \text{has}, 1, \bot\rangle$$

along with a happens-before order that, compared to the linearization order, keeps $\langle \text{has}, 1, \bot\rangle$ unordered w.r.t. $\langle \text{put}, \langle 0, 1\rangle, \top\rangle$ and $\langle \text{put}, \langle 1, 0\rangle, \bot\rangle$, and a visibility projection where the visibility of every put and get includes all the linearization predecessors and the visibility of $\langle \text{has}, 1, \bot\rangle$ consists of $\langle \text{put}, \langle 1, 1\rangle, \top\rangle$ and $\langle \text{put}, \langle 1, 0\rangle, \bot\rangle$. Recall that in the argument $\langle k, v\rangle$ to put operations, the key k precedes value v.

To determine the consistency of individual histories against weak-visibility specifications, we consider adherence of their corresponding abstract executions. Let $h = \langle O, inv, ret, hb\rangle$ be a history and $e = \langle h, lin, vis\rangle$ a complete abstract execution. Then e is *consistent* with a visibility annotation V and read-only predicate R if for each operation $o \in \text{dom}(lin)$ with $inv(o) = \langle m, \_\rangle$, $vis(o)$ is absolute or monotonic, respectively, according to $V(m)$ and R. The *labeling* $\lambda_0\lambda_1 \ldots$ of a total order $o_0 \prec o_1 \prec \ldots$ of complete operations is the sequence of operation labels, i.e., λ_i is the label of o_i. Then e is *consistent* with a sequential specification S when the labeling[9] of $lin \mid (vis(o) \cup \{o\})$ is included in S, for each operation $o \in \text{dom}(lin)$.[10] Finally, we say e is *consistent* with a weak-visibility specification $\langle S, R, V\rangle$ when it is consistent with S, R, and V.

Example 4. The execution in Example 3 is consistent with the weakly-consistent contains-value map W_{m} defined in Example 2.

Remark 1. Consistency models suited for modern software platforms like Java are based on *happens-before* relations which abstract away from *real-time* execution order. Since happens-before, unlike real-time, is not necessarily an *interval order*, the composition

[7] For convenience we rephrase Emmi and Enea [13]'s notion to ignore read-only predecessors.

[8] For readability, we list linearization sequences with operation labels in place of identifiers.

[9] As is standard, adequate labelings of incomplete executions are obtained by completing each linearized yet pending operation with some arbitrarily-chosen return value [18]. It is sufficient that one of these completions be included in the sequential specification.

[10] We consider a simplification from prior work [13]: rather than allowing the observers of a given operation to pretend they see distinct return values, we suppose that all observers agree on return values. While this is more restrictive in principle, it is equivalent for the simple specifications studied in this article.

of linearizations of two distinct objects in the same execution may be cyclic, i.e., not linearizable. Recovering compositionality in this setting is orthogonal to our work of proving consistency against a given model, and is explored elsewhere [11].

The *abstract executions* $E(W)$ of a weak-visibility specification $W = \langle S, R, V \rangle$ include those complete, sequential, and absolute abstract executions derived from sequences of S, i.e., when $s = \lambda_0 \dots \lambda_n \in S$ then each e_s labels each o_i by λ_i, and orders $hb(o_i, o_j)$ iff $i < j$. In addition, when $E(W)$ includes an abstract execution $\langle h, lin, vis \rangle$ with $h = \langle O, inv, ret, hb \rangle$, then $E(W)$ also includes any:

- execution $\langle h', lin, vis \rangle$ such that $h' = \langle O, inv, ret, hb' \rangle$ and $hb' \subseteq hb$; and
- W-consistent execution $\langle h', lin, vis' \rangle$ with $h' = \langle O, inv, ret', hb \rangle$ and $vis' \subseteq vis$.

Note that while *happens-before weakening* $hb' \subseteq hb$ always yields consistent executions, unguarded *visibility weakening* $vis' \subseteq vis$ generally breaks consistency with visibility annotations and sequential specifications: visibilities can become non-monotonic, and return values can change when operations observe fewer operations' effects.

Lemma 1. *The abstract executions $E(W)$ of a specification W are consistent with W.*

Example 5. The abstract executions of W_{m} include the complete, sequential, and absolute abstract execution defined by the following happens-before order

$$\langle \mathrm{put}, \langle 1, 1 \rangle, \top \rangle \; \langle \mathrm{get}, 1, 1 \rangle \; \langle \mathrm{put}, \langle 0, 1 \rangle, \top \rangle \; \langle \mathrm{put}, \langle 1, 0 \rangle, \bot \rangle \; \langle \mathrm{has}, 1, \top \rangle$$

which implies that it also includes one in which just the happens-before order is modified such that $\langle \mathrm{has}, 1, \top \rangle$ becomes unordered w.r.t. $\langle \mathrm{put}, \langle 0, 1 \rangle, \top \rangle$ and $\langle \mathrm{put}, \langle 1, 0 \rangle, \bot \rangle$. Since it includes the latter, it also includes the execution in Example 3 where the visibility of has is weakened which also modifies its return value from $\top$ to $\bot$.

Definition 2. *The* histories *of a weak-visibility specification W are the projections $H(W) = \{h : \langle h, \_, \_ \rangle \in E(W)\}$ of its abstract executions.*

2.2 Consistency against Weak-Visibility Specifications

To define the consistency of implementations against specifications, we leverage a general model of computation to capture the behavior of typical concurrent systems, e.g., including multiprocess and multithreaded systems. A *sequence-labeled transition system* $\langle Q, A, q, \rightarrow \rangle$ is a set Q of states, along with a set A of actions, initial state $q \in Q$ and transition relation $\rightarrow \in Q \times A^* \times Q$. An *execution* is an alternating sequence $\eta = q_0 \vec{a}_0 q_1 \vec{a}_1 \dots q_n$ of states and action sequences starting with $q_0 = q$ such that $q_i \xrightarrow{\vec{a}_i} q_{i+1}$ for each $0 \leq i < n$. The *trace* $\tau \in A^*$ of the execution η is its projection $\vec{a}_0 \vec{a}_1 \dots$ to individual actions.

To capture the histories admitted by a given implementation, we consider sequence-labeled transition systems (SLTSs) which expose actions corresponding to method call, return, and happens-before constraints. We refer to the actions $\mathrm{call}(o, m, x)$, $\mathrm{ret}(o, y)$, and $\mathrm{hb}(o, o')$, for $o, o' \in \mathbb{N}$, $m \in \mathbb{M}$, and $x, y \in \mathbb{X}$, as *the history actions*, and a *history transition system* is an SLTS whose actions include the history actions. We say that an

action over operation identifier o is an *o-action*, and assume that executions are *well formed* in the sense that for a given operation identifier o: at most one call o-action occurs, at most one ret o-action occurs, and no ret nor hb o-actions occur prior to a call o-action. Furthermore, we assume call o-actions are enabled, so long as no prior call o-action has occurred. The *history* of a trace τ is defined inductively by $f_{\mathrm{h}}(h_\emptyset, \tau)$, where $h_\emptyset$ is the empty history, and,

$$\begin{array}{lll} f_{\mathrm{h}}(h, \varepsilon) & = h & g_{\mathrm{h}}(h, \mathrm{call}(o, m, x)) = \langle O \cup \{o\}, inv[o \mapsto \langle m, x\rangle], ret, hb\rangle \\ f_{\mathrm{h}}(h, a\tau) & = f_{\mathrm{h}}(g_{\mathrm{h}}(h, a), \tau) & g_{\mathrm{h}}(h, \mathrm{ret}(o, y)) \quad = \langle O, inv, ret[o \mapsto y], hb\rangle \\ f_{\mathrm{h}}(h, \tilde{a}\tau) & = f_{\mathrm{h}}(h, \tau) & g_{\mathrm{h}}(h, \mathrm{hb}(o, o')) \quad = \langle O, inv, ret, hb \cup \langle o, o'\rangle\rangle \end{array}$$

where $h = \langle O, inv, ret, hb\rangle$, and a is a call, ret, or hb action, and $\tilde{a}$ is not. An *implementation* I is a history transition system, and the *histories* $H(I)$ of I are those of its traces. Finally, we define consistency against specifications via history containment.

Definition 3. *Implementation I is* consistent *with specification W iff $H(I) \subseteq H(W)$.*

3 Establishing Consistency with Forward Simulation

To obtain a consistency proof strategy, we more closely relate implementations to specifications via their admitted abstract executions. To capture the abstract executions admitted by a given implementation, we consider SLTSs which expose not only history-related actions, but also actions witnessing linearization and visibility. We refer to the actions $\mathrm{lin}(o)$ and $\mathrm{vis}(o, o')$ for $o, o' \in \mathbb{N}$, along with the history actions, as *the abstract-execution actions*, and an *abstract-execution transition system* (AETS) is an SLTS whose actions include the abstract-execution actions. Extending the corresponding notion from history transition systems, we assume that executions are *well formed* in the sense that for a given operation identifier o: at most one lin o-action occurs, and no lin or vis o-actions occur prior to a call o-action. The *abstract execution* of a trace τ is defined inductively by $f_{\mathrm{e}}(e_\emptyset, \tau)$, where $e_\emptyset = \langle h_\emptyset, \emptyset, \emptyset\rangle$ is the empty execution, and,

$$\begin{array}{lll} f_{\mathrm{e}}(e, \varepsilon) & = e & g_{\mathrm{e}}(e, \hat{a}) \quad\quad = \langle g_{\mathrm{h}}(h), lin, vis\rangle \\ f_{\mathrm{e}}(e, a\tau) & = f_{\mathrm{e}}(g_{\mathrm{e}}(e, a), \tau) & g_{\mathrm{e}}(e, \mathrm{lin}(o)) \quad = \langle h, lin \cup \{\langle o', o\rangle : o' \in lin\}, vis\rangle \\ f_{\mathrm{e}}(e, \tilde{a}\tau) & = f_{\mathrm{e}}(e, \tau) & g_{\mathrm{e}}(e, \mathrm{vis}(o, o')) = \langle h, lin, vis \cup \{\langle o, o'\rangle\}\rangle \end{array}$$

where $e = \langle h, lin, vis\rangle$, and a is a call, ret, hb, lin, or vis action, $\tilde{a}$ is not, and $\hat{a}$ is a call, ret, or hb action. A *witnessing implementation* I is an abstract-execution transition system, and the *abstract executions* $E(I)$ of I are those of its traces.

We adopt forward simulation [25] for proving consistency against weak-visibility specifications. Formally, a *simulation relation* from one system $\Sigma_1 = \langle Q_1, A_1, \chi_1, \rightarrow_1\rangle$ to another $\Sigma_2 = \langle Q_2, A_2, \chi_2, \rightarrow_2\rangle$ is a binary relation $R \subseteq Q_1 \times Q_2$ such that initial states are related, $R(\chi_1, \chi_2)$, and: for any pair of related states $R(q_1, q_2)$ and source-system transition $q_1 \xrightarrow{\vec{a}_1}_1 q_1'$, there exists a target-system transition $q_2 \xrightarrow{\vec{a}_2}_2 q_2'$ to related states, i.e., $R(q_1', q_2')$, over common actions, i.e., $(\vec{a}_1 \mid A_2) = (\vec{a}_2 \mid A_1)$. We say Σ_2 *simulates* Σ_1 and write $\Sigma_1 \sqsubseteq \Sigma_2$ when a simulation relation from Σ_1 to Σ_2 exists.

We derive transition systems to model consistency specifications in simulation. The following lemma establishes the soundness and completeness of this substitution, and the subsequent theorem asserts the soundness of the simulation-based proof strategy.

Definition 4. *The* transition system $[\![W]\!]_{\mathrm{s}}$ *of a weak-visibility specification* W *is the AETS whose actions are the abstract execution actions, whose states are abstract executions, whose initial state is the empty execution, and whose transitions include* $e_1 \xrightarrow{\vec{a}} e_2$ *iff* $f_{\mathrm{e}}(e_1, \vec{a}) = e_2$ *and* e_2 *is consistent with* W.

Lemma 2. *A weak-visibility spec. and its transition system have identical histories.*

Theorem 1. *A witnessing implementation* I *is consistent with a weak-visibility specification* W *if the transition system* $[\![W]\!]_{\mathrm{s}}$ *of* W *simulates* I.

Our notion of simulation is in some sense *complete* when the sequential specification S of a weak-consistency specification $W = \langle S, R, V \rangle$ is *return-value deterministic*, i.e., there is a single label $\langle m, x, y \rangle$ such that $\vec{\lambda} \cdot \langle m, x, y \rangle \in S$ for any method m, argument-value x, and admitted sequence $\vec{\lambda} \in S$. In particular, $[\![W]\!]_{\mathrm{s}}$ simulates any witnessing implementation I whose abstract executions $E(I)$ are included in $E([\![W]\!]_{\mathrm{s}})$.[11] This completeness, however, extends only to inclusion of abstract executions, and not all the way to consistency, since consistency is defined on histories, and any given operation's return value is not completely determined by the other operation labels and happens-before relation of a given history: return values generally depend on linearization order and visibility as well. Nevertheless, sequential specifications typically are return-value deterministic, and we have used simulation to prove consistency of Java-inspired weakly-consistent objects.

Establishing simulation for an implementation is also helpful when reasoning about clients of a concurrent object. One can use the specification in place of the implementation and encode the client invariants using the abstract execution of the specification in order to prove client properties, following Sergey et al.'s approach [35].

3.1 Reducing Consistency to Safety Verification

Proving simulation between an implementation and its specification can generally be achieved via product construction: complete the transition system of the specification, replacing non-enabled transitions with error-state transitions; then ensure the synchronized product of implementation and completed-specification transition systems is *safe*, i.e., no error state is reachable. Assuming that the individual transition systems are safe, then the product system is safe *iff* the specification simulates the implementation. This reduction to safety verification is also generally applicable to implementation and specification programs, though we limit our formalization to their underlying transition systems for simplicity. By the upcoming Corollary 1, such reductions enable consistency verification with existing safety verification tools.

3.2 Verifying Implementations

While Theorem 1 establishes forward simulation as a strategy for proving the consistency of implementations against weak-visibility specifications, its application to

[11] This is a consequence of a generic result stating that the set of traces of an LTS A_1 is included in the set of traces of an LTS A_2 iff A_2 simulates A_1, provided that A_2 is deterministic [25].

real-world implementations requires program-level mechanisms to signal the underlying AETS lin and vis actions. To apply forward simulation, we thus develop a notion of programs whose commands include such mechanisms.

This section illustrates a toy programming language with AETS semantics which provides these mechanisms. The key features are the lin and vis program commands, which emit linearization and visibility actions for the currently-executing operation, along with load, store, and cas (compare-and-swap) commands, which record and return the set of operation identifiers having written to each memory cell. Such augmented memory commands allow programs to obtain handles to the operations whose effects it has observed, in order to signal the corresponding vis actions.

While one can develop similar mechanisms for languages with any underlying memory model, the toy language presented here assumes a sequentially-consistent memory. Note that the assumption of sequentially-consistent memory operations is practically without loss of generality for Java 8's concurrent collections since they are designed to be data-race free — their anomalies arise not from weak-memory semantics, but from non-atomic operations spanning several memory cells.

For generality, we assume abstract notions of commands and memory, using κ, μ, ℓ, and M respectively to denote a *program command*, *memory command*, *local state*, and *global memory*. So that operations can assert their visibilities, we consider memory which stores, and returns upon access, the identifier(s) of operations which previously accessed a given cell. A *program* $P = \langle \mathsf{init}, \mathsf{cmd}, \mathsf{idle}, \mathsf{done} \rangle$ consists of an $\mathsf{init}(m, x) = \ell$ function mapping method name m and argument values x to local state ℓ, along with a $\mathsf{cmd}(\ell) = \kappa$ function mapping local state ℓ to program command κ, and $\mathsf{idle}(\ell)$ and $\mathsf{done}(\ell)$ predicates on local states ℓ. Intuitively, identifying local states with threads, the idle predicate indicates whether a thread is outside of atomic sections, and subject to interference from other threads; meanwhile the done predicate indicates whether whether a thread has terminated.

The *denotation* of a memory command μ is a function $[\![\mu]\!]_{\mathrm{m}}$ from global memory M_1, argument value x, and operation o to a tuple $[\![\mu]\!]_{\mathrm{m}}(M_1, x, o) = \langle M_2, y \rangle$ consisting of a global memory M_2, along with a return value y.

Example 6. A sequentially-consistent memory system which records the set of operations to access each location can be captured by mapping addresses x to value and operation-set pairs $M(x) = \langle y, O \rangle$, along with three memory commands:

$$[\![\mathsf{load}]\!]_{\mathrm{m}}(M, x, \_) = \langle M, M(x) \rangle$$

$$[\![\mathsf{store}]\!]_{\mathrm{m}}(M, xy, o) = \langle M[x \mapsto \langle y, M(x)_1 \cup \{o\} \rangle], \varepsilon \rangle$$

$$[\![\mathsf{cas}]\!]_{\mathrm{m}}(M, xyz, o) = \begin{cases} \langle M[x \mapsto \langle z, M(x)_1 \cup \{o\} \rangle], \langle \mathit{true}, M(x)_1 \rangle \rangle & \text{if } M(x)_0 = y \\ \langle M, \langle \mathit{false}, M(x)_1 \rangle \rangle & \text{if } M(x)_0 \neq y \end{cases}$$

where the compare-and-swap (CAS) operation stores value z at address x and returns true when y was previously stored, and otherwise returns false.

The *denotation* of a program command κ is a function $[\![\kappa]\!]_{\mathrm{c}}$ from local state ℓ_1 to a tuple $[\![\kappa]\!]_{\mathrm{c}}(\ell_1) = \langle \mu, x, f \rangle$ consisting of a memory command μ and argument value x,

and a update continuation f mapping the memory command's return value y to a pair $f(y) = \langle \ell_2, \alpha \rangle$, where ℓ_2 is an updated local state, and α maps an operation o to an LTS action $\alpha(o)$. We assume the denotation $[\![\texttt{ret}\ x]\!]_{\mathrm{c}}(\ell_1) = \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \ell_2, \lambda o.\mathrm{ret}(z)\rangle\rangle$ of the `ret` command yields a local state ℓ_2 with $\mathsf{done}(\ell_2)$ without executing memory commands, and outputs a corresponding LTS ret action.

Example 7. A simple goto language over variables $\mathtt{a}, \mathtt{b}, \ldots$ for the memory system of Example 6 would include the following commands:

$$
\begin{aligned}
[\![\texttt{goto a}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{jump}(\ell, \ell(a)), \lambda o.\varepsilon\rangle\rangle \\
[\![\texttt{assume a}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{next}(\ell), \lambda o.\varepsilon\rangle\rangle \text{ if } \ell(\mathtt{a}) \neq 0 \\
[\![\mathtt{b}, \mathtt{c} = \texttt{load(a)}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{load}, \ell(\mathtt{a}), \lambda y_1, y_2.\langle \mathit{next}(\ell[\mathtt{b} \mapsto y_1][\mathtt{c} \mapsto y_2]), \lambda o.\varepsilon\rangle\rangle \\
[\![\texttt{store(a, b)}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{store}, \ell(\mathtt{a})\ell(\mathtt{b}), \lambda y.\langle \mathit{next}(\ell), \lambda o.\varepsilon\rangle\rangle \\
[\![\mathtt{d}, \mathtt{e} = \texttt{cas(a, b, c)}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{cas}, \ell(\mathtt{a})\ell(\mathtt{b})\ell(\mathtt{c}), \lambda y_1, y_2.\langle \mathit{next}(\ell[\mathtt{d} \mapsto y_1][\mathtt{e} \mapsto y_2]), \lambda o.\varepsilon\rangle\rangle
\end{aligned}
$$

where the *jump* and *next* functions update a program counter, and the load command stores the operation identifier returned from the corresponding memory commands. Linearization and visibility actions are captured as program commands as follows:

$$
\begin{aligned}
[\![\texttt{lin}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{next}(\ell), \lambda o.\mathrm{lin}(o)\rangle\rangle \\
[\![\texttt{vis(a)}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{next}(\ell), \lambda o.\mathrm{vis}(o, \ell(\mathtt{a}))\rangle\rangle
\end{aligned}
$$

Atomic sections can be captured with a `lock` variable and a pair of program commands,

$$
\begin{aligned}
[\![\texttt{begin}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{next}(\ell[\texttt{lock} \mapsto \mathit{true}]), \lambda o.\varepsilon\rangle\rangle \\
[\![\texttt{end}]\!]_{\mathrm{c}}(\ell) &= \langle \mathsf{nop}, \varepsilon, \lambda y.\langle \mathit{next}(\ell[\texttt{lock} \mapsto \mathit{false}]), \lambda o.\varepsilon\rangle\rangle
\end{aligned}
$$

such that idle states are identified by not holding the lock, i.e., $\mathsf{idle}(\ell) = \neg\ell(\texttt{lock})$, as in the initial state $\mathsf{init}(m, x)(\texttt{lock}) = \mathit{false}$.

Figure 1 lists the semantics $[\![P]\!]_{\mathrm{p}}$ of a program P as an abstract-execution transition system. The states $\langle M, L\rangle$ of $[\![P]\!]_{\mathrm{p}}$ include a global memory M, along with a partial function L from operation identifiers o to local states $L(o)$; the initial state is $\langle M_\emptyset, \emptyset\rangle$, where $M_\emptyset$ is an initial memory state. The transitions for call and hb actions are enabled independently of implementation state, since they are dictated by implementations' environments. Although we do not explicitly model client programs and platforms here, in reality, client programs dictate call actions, and platforms, driven by client programs, dictate hb actions; for example, a client which acquires the lock released after operation o_1, before invoking operation o_2, is generally ensured by its platform that o_1 happens before o_2. The transitions for all other actions are dictated by implementation commands. While the ret, lin, and vis commands generate their corresponding LTS actions, all other commands generate ε transitions.

Each atomic $\xrightarrow{\vec{a}}$ step of the AETS underlying a given program is built from a sequence of $\rightsquigarrow$ steps for the individual program commands in an atomic section. Individual program commands essentially execute one small $\rightsquigarrow$ step from shared memory and local state $\langle M_1, \ell_1\rangle$ to $\langle M_2, \ell_2\rangle$, invoking memory command μ with

$$\frac{o \notin \mathrm{dom}(L) \qquad \ell = \mathsf{init}(m,x)}{\langle M, L\rangle \xrightarrow{\mathrm{call}(o,m,x)} \langle M, L[o \mapsto \ell]\rangle} \qquad \frac{\mathsf{done}(L(o_1)) \qquad o_2 \notin \mathrm{dom}(L)}{\langle M, L\rangle \xrightarrow{\mathrm{hb}(o_1,o_2)} \langle M, L\rangle}$$

$$\frac{\langle M_1, \ell_1, o, \varepsilon\rangle \rightsquigarrow^* \langle M_2, \ell_2, o, \vec{a}\rangle \qquad \mathsf{idle}(\ell_2)}{\langle M_1, L[o \mapsto \ell_1]\rangle \xrightarrow{\vec{a}} \langle M_2, L[o \mapsto \ell_2]\rangle}$$

$$\frac{\begin{array}{c}\mathsf{cmd}(\ell_1) = \kappa \qquad [\![\kappa]\!]_{\mathrm{c}}(\ell_1) = \langle \mu, x, f\rangle \\ [\![\mu]\!]_{\mathrm{m}}(M_1, x, o) = \langle M_2, y\rangle \qquad f(y) = \langle \ell_2, \alpha\rangle\end{array}}{\langle M_1, \ell_1, o, \vec{a}\rangle \rightsquigarrow \langle M_2, \ell_2, o, \vec{a} \cdot \alpha(o)\rangle}$$

Fig. 1. The semantics of program $P = \langle \mathsf{init}, \mathsf{cmd}, \mathsf{idle}, \mathsf{done}\rangle$ as an abstract-execution transition system, where $[\![\cdot]\!]_{\mathrm{c}}$ and $[\![\cdot]\!]_{\mathrm{m}}$ are the denotations of program and memory commands, respectively.

argument x, and emitting action $\alpha(o)$. Besides its effect on shared memory, each step uses the result $\langle M_2, y\rangle$ of memory command μ to update local state and emit an action using the continuation f, i.e., $f(y) = \langle \ell_2, \alpha\rangle$. Commands which do not access memory are modeled by a no-op memory commands. We define the consistency of programs by reduction to their transition systems.

Definition 5. *A program P is* consistent *with a specification iff its semantics $[\![P]\!]_{\mathrm{p}}$ is.*

Thus the consistency of P with W amounts to the inclusion of $[\![P]\!]_{\mathrm{p}}$'s histories in W's. The following corollary of Theorem 1 follows directly by Definition 5, and immediately yields a program verification strategy: validate a simulation relation from the states of $[\![P]\!]_{\mathrm{p}}$ to the states of $[\![W]\!]_{\mathrm{s}}$ such that each command of P is simulated by a step of $[\![W]\!]_{\mathrm{s}}$.

Corollary 1. *A program P is consistent with specification W if $[\![W]\!]_{\mathrm{s}}$ simulates $[\![P]\!]_{\mathrm{p}}$.*

4 Proof Methodology

In this section we develop a systematic means to annotating concurrent objects for relaxed-visibility simulation proofs. Besides leveraging an auxiliary memory system which tags memory accesses with the operation identifiers which wrote read values (see §3.2), annotations signal linearization points with lin commands, and indicate visibility of other operations with vis commands. As in previous works [3, 37, 2, 18] we assume linearization points are given, and focus on visibility-related annotations.

As we focus on data-race free implementations (e.g., Java 8's concurrent collections) for which sequential consistency is sound, it can be assumed without loss of generality that the happens-before order is exactly the *returns-before* order between operations, which orders two operations o_1 and o_2 iff the return action of o_1 occurs in real-time before the call action of o_2. This assumption allows to guarantee that linearizations are consistent with happens-before just by ensuring that the linearization point of each operation occurs in between its call and return action (like in standard linearizability).

```
var table: array of T;

procedure absolute put(k: int, v: T) {
  atomic {
    store(table[k], v);
    vis(getLin());
    lin();
  }
}

procedure absolute get(k: int) {
  atomic{
    v, O = load(table[k]);
    vis(getLin());
    lin();
  }
  return v;
}

procedure monotonic has(v: T)
  vis(getModLin());
{
  store(k, 0);
  while (k < table.length) {
    atomic{
      tv, O = load(table[k]);
      vis(O ∩ getModLin());
    }
    if (tv = v) then {
      lin();
      return true;
    }
    inc(k);
  }
  lin();
  return false;
}
```

Fig. 2. An implementation I_{chm} modeling Java's concurrent hash map. The command `inc(k)` increments counter `k`, and commands within `atomic` $\{\ldots\}$ are collectively atomic.

It is without loss of generality because the clients of such implementations can use auxiliary variables to impose synchronization order constraints between every two operations ordered by returns-before, e.g., writing a variable after each operation returns which is read before each other operation is called (under sequential consistency, every write happens-before every other read which reads the written value).

We illustrate our methodology with the key-value map implementation I_{chm} of Figure 2, which models Java's concurrent hash map. The lines marked in blue and red represent linearization/visibility commands added by the instrumentation that will be described below. Key-value pairs are stored in an array `table` indexed by keys. The implementation of `put` and `get` are obvious while the implementation of `has` returns true iff the input value is associated to some key consists of a while loop traversing the array and searching for the input value. To simplify the exposition, the shared memory reads and writes are already adapted to the memory system described in Section 3.2 (essentially, this consists in adding new variables storing the set of operation identifiers returned by a shared memory read). While `put` and `get` are obviously linearizable, `has` is weakly consistent, with monotonic visibility. For instance, given the two thread program $\{\texttt{get(1)};\texttt{has(1)}\} \parallel \{\texttt{put(1,1)};\texttt{put(0,1)};\texttt{put(1,0)}\}$ it is possible that `get(1)` returns 1 while `has(1)` returns false. This is possible in an interleaving where `has` reads `table[0]` before `put(0,1)` writes into it (observing the initial value 0), and `table[1]` after `put(1,0)` writes into it (observing value 0 as well). The only abstract execution consistent with the weakly-consistent contains-value map W_{m} (Example 2) which justifies these return values is given in Example 3. We show that this implementation is consistent with a simplification of the contains-value map W_{m}, without remove key operations, and where put operations return no value.

Given an implementation I, let $\mathcal{L}(I)$ be an instrumentation of I with program commands `lin()` emitting linearization actions. The execution of `lin()` in the context of an operation with identifier o emits a linearization action $\text{lin}(o)$. We assume that $\mathcal{L}(I)$ leads to well-formed executions (e.g., at most one linearization action per operation).

Example 8. For the implementation in Figure 2, the linearization commands of `put` and `get` are executed atomically with the store to `table[k]` in `put` and the load of `table[k]` in `get`, respectively. The linearization command of `has` is executed at any point after observing the input value `v` or after exiting the loop, but before the return. The two choices correspond to different return values and only one of them will be executed during an invocation.

Given an instrumentation $\mathcal{L}(I)$, a visibility annotation V for I's methods, and a read-only predicate R, we define a witnessing implementation $\mathcal{V}(\mathcal{L}(I))$ according to a generic heuristic that depends only on V and R. This definition uses a program command `getLin()` which returns the set of operations in the current linearization sequence.[12] The current linearization sequence is stored in a history variable which is updated with every linearization action by appending the corresponding operation identifier. For readability, we leave this history variable implicit and omit the corresponding updates. As syntactic sugar, we use a command `getModLin()` which returns the set of *modifiers* (non read-only operations) in the current linearization sequence. To represent visibility actions, we use program commands `vis(A)` where `A` is a set of operation identifiers. The execution of `vis(A)` in the context of an operation with identifier o emits the set of visibility actions $\mathrm{vis}(o, o')$ for every operation $o' \in$ `A`.

Therefore, $\mathcal{V}(\mathcal{L}(I))$ extends the instrumentation $\mathcal{L}(I)$ with commands generating visibility actions as follows:

- for absolute methods, each linearization command is preceded by `vis(getLin())` which ensures that the visibility of an invocation includes all the predecessors in linearization order. This is executed atomically with `lin()`.
- for monotonic methods, the call action is followed by `vis(getModLin())` (and executed atomically with this command) which ensures that the visibility of each invocation is monotonic, and every read of a shared variable which has been written by a set of operations O is preceded by `vis(O ∩ getModLin())` (and executed atomically with this command). The latter is needed so that the visibility of such an invocation contains enough operations to explain its return value (the visibility command attached to call actions is enough to ensure monotonic visibilities).

Example 9. The blue lines in Figure 2 demonstrate the visibility commands added by the instrumentation $\mathcal{V}(\cdot)$ to the key-value map in Figure 2 (in this case, the modifiers are `put` operations). The first visibility command in `has` precedes the procedure body to emphasize the fact that it is executed *atomically* with the procedure call. Also, note that the read of the array `table` is the only shared memory read in `has`.

Theorem 2. *The abstract executions of the witnessing implementation $\mathcal{V}(\mathcal{L}(I))$ are consistent with V and R.*

Proof. Let $\langle h, lin, vis\rangle$ be the abstract execution of a trace τ of $\mathcal{V}(\mathcal{L}(I))$, and let o be an invocation in h of a monotonic method (w.r.t. V). By the definition of $\mathcal{V}$, the call action of o is *immediately* followed in τ by a sequence of visibility actions $\mathrm{vis}(o, o')$

[12] We rely on retrieving the identifiers of currently-linearized operations. More complex proofs may also require inspecting, e.g., operation labels and happens-before relationships.

for every modifier o' which has been already linearized. Therefore, any operation which has returned before o (i.e., happens-before o) has already been linearized and it will necessarily have a smaller visibility (w.r.t. set inclusion) because the linearization sequence is modified only by appending new operations. The instrumentation of shared memory reads may add more visibility actions $\mathrm{vis}(o, \_)$ but this preserves the monotonicity status of o's visibility. The case of absolute methods is obvious. □

The consistency of the abstract executions of $\mathcal{V}(\mathcal{L}(I))$ with a given sequential specification S, which completes the proof of consistency with a weak-visibility specification $W = \langle S, R, V\rangle$, can be proved by showing that the transition system $[\![W]\!]_{\mathrm{s}}$ of W simulates $\mathcal{V}(\mathcal{L}(I))$ (Theorem 1). Defining a simulation relation between the two systems is in some part implementation specific, and in the following we demonstrate it for the key-value map implementation $\mathcal{V}(\mathcal{L}(I_{\mathrm{chm}}))$.

We show that $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$ simulates implementation I_{chm}. A state of I_{chm} in Figure 2 is a valuation of `table` and the history variable `lin` storing the current linearization sequence, and a valuation of the local variables for each active operation. Let $ops(q)$ denote the set of operations which are active in an implementation state q. Also, for a `has` operation $o \in ops(q)$, let $index(o)$ be the maximal index k of the array `table` such that o has already read `table`$[k]$ and `table`$[k] \neq$ `v`. We assume $index(o) = -1$ if o did not read any array cell.

Definition 6. *Let R_{chm} be a relation which associates every implementation state q with a state of $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$, i.e., an $\langle S, R, V\rangle$-consistent abstract execution $e = \langle h, lin, vis\rangle$ with $h = \langle O, inv, ret, hb\rangle$, such that:*

1. *O is the set of identifiers occurring in $ops(q)$ or the history variable* `lin`*,*
2. *for each operation $o \in ops(q)$, $inv(o)$ is defined according to its local state, $ret(o)$ is undefined, and o is maximal in the happens-before order hb,*
3. *the value of the history variable* `lin` *in q equals the linearization sequence lin,*
4. *every invocation $o \in ops(q)$ of an absolute method (*`put` *or* `get`*) has absolute visibility if linearized, otherwise, its visibility is empty,*
5. `table` *is the array obtained by executing the sequence of operations lin,*
6. *for every linearized* `get(k)` *operation $o \in ops(q)$, the* `put(k,_)` *operation in $vis(o)$ which occurs last in lin writes* `v` *to key* `k`*, where* `v` *is the local variable of o,*
7. *for every* `has` *operation $o \in ops(q)$, $vis(o)$ consists of:*
   - *all the* `put` *operations o' which returned before o was invoked,*
   - *for each $i \leq index(o)$, all the* `put(`i`,_)` *operations from a prefix of lin that wrote a value* different *from* `v`*,*
   - *all the* `put(`$index(o)+1$`,_)` *operations from a prefix of lin that ends with a* `put(`$index(o)+1$`,v)` *operation, provided that* `tv` $=$ `v`*.*

   Above, the linearization prefix associated to an index $j_1 < j_2$ should be a prefix of the one associated to j_2.

A large part of this definition is applicable to any implementation, only points (5), (6), and (7) being specific to the implementation we consider. The points (6) and (7) ensure that the return values of operations are consistent with S and mimic the effect of the `vis` commands from Figure 2.

Theorem 3. *R_{chm} is a simulation relation from $\mathcal{V}(\mathcal{L}(I_{\mathrm{chm}}))$ to $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$.*

5 Implementation and Evaluation

In this section we effectuate our methodology by verifying two weakly-consistent concurrent objects: Java's `ConcurrentHashMap` and `ConcurrentLinkedQueue`.[13] We use an off-the-shelf deductive verification tool called CIVL [16], though any concurrent program verifier could suffice. We chose CIVL because comparable verifiers either require a manual encoding of the concurrency reasoning (e.g. Dafny or Viper) which can be error-prone, or require cumbersome reasoning about interleavings of thread-local histories (e.g. VerCors). An additional benefit of CIVL is that it directly proves simulation, thereby tying the mechanized proofs to our theoretical development. Our proofs assume no bound on the number of threads or the size of the memory.

Our use of CIVL imposes two restrictions on the implementations we can verify. First, CIVL uses the Owicki-Gries method [29] to verify concurrent programs. These methods are unsound for weak memory models [22], so CIVL, and hence our proofs, assume a sequentially-consistent memory model. Second, CIVL's strategy for building the simulation relation requires implementations to have statically-known linearization points because it checks that there exists exactly one atomic section in each code path where the global state is modified, and this modification is simulated by the specification.

Given these restrictions, we can simplify our proof strategy of forward refinement by factoring the simulations we construct through an atomic version of the specification transition system. This atomic specification is obtained from the specification AETS $[\![W]\!]_{\mathrm{s}}$ by restricting the interleavings between its transitions.

Definition 7. *The* atomic transition system *of a specification* W *is the AETS* $[\![W]\!]_{\mathrm{a}} = \langle Q, A, q, \rightarrow_a \rangle$*, where* $[\![W]\!]_{\mathrm{s}} = \langle Q, A, q, \rightarrow \rangle$ *is the AETS of* W *and* $e_1 \xrightarrow{\vec{a}}_a e_2$ *if and only if* $e_1 \xrightarrow{\vec{a}} e_2$ *and* $\vec{a} \in \{\mathrm{call}(o, m, x)\} \cup \{\mathrm{ret}(o, y)\} \cup \{\mathrm{hb}(o, o')\} \cup \{\vec{a_1}\ \mathrm{lin}(o) : \vec{a_1} \in \{\mathrm{vis}(o, \_)\}^*\}$.

Note that the language of $[\![W]\!]_{\mathrm{a}}$ is included in the language of $[\![W]\!]_{\mathrm{s}}$ and simulation proofs towards $[\![W]\!]_{\mathrm{a}}$ apply to $[\![W]\!]_{\mathrm{s}}$ as well.

Our CIVL proofs show that there is a simulation from an implementation to its atomic specification, which is encoded as a program whose state consists of the components of an abstract execution, i.e., $\langle O, inv, ret, hb, lin, vis \rangle$. These were encoded as maps from operation identifiers to values, sequences of operation identifiers, and maps from operation identifiers to sets of operation identifiers respectively. Our axiomatization of sequences and sets were adapted from those used by the Dafny verifier [23]. For each method in $\mathbb{M}$, we defined atomic procedures corresponding to call actions, return actions, and combined visibility and linearization actions in order to obtain exactly the atomic transitions of $[\![W]\!]_{\mathrm{a}}$.

It is challenging to encode Java implementations faithfully in CIVL, as the latter's input programming language is a basic imperative language lacking many Java features. Most notable among these is dynamic memory allocation on the heap, used by almost all of the concurrent data structure implementations. As CIVL is a first-order prover, we needed an encoding of the heap that lets us perform reachability reasoning on the

[13] Our verified implementations are open source, and available at: https://github.com/siddharth-krishna/weak-consistency-proofs.

heap. We adapted the first-order theory of reachability and footprint sets from the GRASShopper verifier [30] for dynamically allocated data structures. This fragment is decidable, but relies on local theory extensions [36], which we implemented by using the trigger mechanism of the underlying SMT solver [27, 15] to ensure that quantified axioms were only instantiated for program expressions. For instance, here is the "cycle" axiom that says that if a node `x` has a field `f[x]` that points to itself, then any `y` that it can reach via that field (encoded using the between predicate `Btwn(f, x, y, y)`) must be equal to `x`:

```
axiom (forall f: [Ref]Ref, x: Ref, y:Ref :: {known(x), known(y)}
       f[x] == x && Btwn(f, x, y, y) ==> x == y);
```

We use the trigger `known(x), known(y)` (`known` is a dummy function that maps every reference to true) and introduce `known(t)` terms in our programs for every term `t` of type `Ref` (for instance, by adding `assert known(t)` to the point of the program where `t` is introduced). This ensures that the cycle axiom is only instantiated for terms that appear in the program, and not for terms that are generated by instantations of axioms (like `f[x]` in the cycle axiom). This process was key to keeping the verification time manageable.

Since we consider fine-grained concurrent implementations, we also needed to reason about interference by other threads and show thread safety. CIVL provides Owicki-Gries [29] style thread-modular reasoning, by means of demarcating atomic blocks and providing preconditions for each block that are checked for stability under all possible modifications by other threads. One of the consequences of this is that these annotations can only talk about the local state of a thread and the shared global state, but not other threads. To encode facts such as distinctness of operation identifiers and ownership of unreachable nodes (e.g. newly allocated nodes) in the shared heap, we use CIVL's linear type system [40].

For instance, the proof of the `push` method needs to make assertions about the value of the newly-allocated node `x`. These assertions would not be stable under interference of other threads if we didn't have a way of specifying that the address of the new node is known only by the `push` thread. We encode this knowledge by marking the type of the variable `x` as linear – this tells CIVL that all values of `x` across all threads are distinct, which is sufficient for the proof. CIVL ensures soundness by making sure that linear variables are not duplicated (for instance, they cannot be passed to another method and then used afterwards).

We evaluate our proof methodology by considering models of two of Java's weakly-consistent concurrent objects.

Concurrent Hash Map One is the `ConcurrentHashMap` implementation of the Map ADT, consisting of absolute `put` and `get` methods and a monotonic `has` method that follows the algortihm given in Figure 2. For simplicity, we assume here that keys are integers and the hash function is identity, but note that the proof of monotonicity of `has` is not affected by these assumptions.[14]

[14] Our CIVL implementation assumes the hash function is injective to avoid reasoning about the dynamic bucket-list needed to resolve hash collisions. While such reasoning is possible within

Module	Code	Proof	Total	Time (s)
Sets and Sequences	-	85	85	-
Executions and Consistency	-	30	30	-
Heap and Reachability	-	35	35	-
Map ADT	51	34	85	-
Array-map implementation	138	175	313	6
Queue ADT	50	22	72	-
Linked Queue implementation	280	325	605	13

Fig. 3. Case study detail: for each object we show lines of code, lines of proof, total lines, and verification time in seconds. We also list common definitions and axiomatizations separately.

CIVL can construct a simulation relation equivalent to the one defined in Definition 6 automatically, given an inductive invariant that relates the state of the implementation to the abstract execution. A first attempt at an invariant might be that the value stored at `table[k]` for every key `k` is the same as the value returned by adding a `get` operation on `k` by the specification AETS. This invariant is sufficient for CIVL to prove that the return value of the absolute methods (`put` and `get`) is consistent with the specification.

However, it is not enough to show that the return value of the monotonic `has` method is consistent with its visibility. This is because our proof technique constructs a visibility set for `has` by taking the union of the memory tags (the set of operations that wrote to each memory location) of each table entry it reads, but without additional invariants this visibility set could entail a different return value. We thus strengthen the invariant to say that `tableTags[k]`, the memory tags associated with hash table entry `k`, is exactly the set of linearized `put` operations with key `k`. A consequence of this is that the abstract state encoded by `tableTags[k]` has the same value for key `k` as the value stored at `table[k]`. CIVL can then prove, given the following loop invariant, that the value returned by `has` is consistent with its visibility set.

```
(forall i: int :: 0 <= i && i < k ==> Map.ofVis(my_vis, lin)[i] != v)
```

This loop invariant says that among the entries scanned thus far, the abstract map given by the projection of `lin` to the current operation's visibility `my_vis` does not include value `v`.

Concurrent Linked Queue Our second case study is the `ConcurrentLinkedQueue` implementation of the Queue ADT, consisting of absolute `push` and `pop` methods and a monotonic `size` method that traverses the queue from head to tail without any locks and returns the number of nodes it sees (see Figure 4 for the full code). We again model the core algorithm (the Michael-Scott queue [26]) and omit some of Java's optimizations, for instance to speed up garbage collection by setting the `next` field of popped nodes to themselves, or setting the values of nodes to `null` when popping values.

The invariants needed to verify the absolute methods are a straightforward combination of structural invariants (e.g. that the queue is composed of a linked list from the head to null, with the tail being a member of this list) and a relation between the

CIVL, see our queue case study, this issue is orthogonal to the weak-consistency reasoning that we study here.

```
var head, tail: Ref;  struct Node { var data: K; var next: Ref; }

procedure absolute push(k: K) {
  x = new Node(k, null);
  while (true) {
    t, _ = load(tail);
    tn, _ = load(tail.next);
    if (tn == null) {
      atomic {
        b, _ = cas(t.next, tn, x);
        if (b) {
          vis(getLin());
          lin();
        }
      }
      if (b) then break;

    } else {
      b, _ = cas(tail, t, tn);
    }
  }
}

procedure absolute pop() {
  while (true) {
    h, _ = load(head);
    t, _ = load(tail);
    hn, _ = load(h.next);
    if (h != t) {
      k, _ = load(hn.data);
      atomic {
        b, _ = cas(head, h, hn);
        if (b) {
          vis(getLin());
          lin();
        }
      }
      if (b) then return k;
    }
  }
}

procedure monotonic size()
  vis(getModLin());
{
  store(s, 0);
  c, _ = load(head);
  atomic {
    cn, 0 = load(c.next);
    vis(0 ∩ getModLin());
  }
  while (cn != null) {
    inc(s);
    c = cn;
    atomic {
      cn, 0 = load(c.next);
      vis(0 ∩ getModLin());
    }
  }
  lin();
  return s;
}
```

Fig. 4. The simplified implementation of Java's `ConcurrentLinkedQueue` that we verify.

abstract and concrete states. Once again, we need to strengthen this invariant in order to verify the monotonic `size` method, because otherwise we cannot prove that the visibility set we construct (by taking the union of the memory tags of nodes in the list during traversal) justifies the return value.

The key additional invariant is that the memory tags for the next field of each node (denoted `x.nextTags` for each node `x`) in the queue contain the operation label of the operation that pushed the next node into the queue (if it exists). Further, the sequence of push operations in `lin` are exactly the operations in the `nextTags` field of nodes in the queue, and in the order they are present in the queue.

Figure 5 shows a simplified version of the CIVL encoding of these invariants. In it, we use the following auxiliary variables in order to avoid quantifier alternation: `nextInvoc` maps nodes to the operation label (type `Invoc` in CIVL) contained in the `nextTags` field; `nextRef` maps operations to the nodes whose `nextTags` field contains them, i.e. it is the inverse of `nextInvoc`; and `absRefs` maps the index of the abstract queue (represented as a mathematical sequence) to the corresponding concrete heap node. We omit the triggers and `known` predicates for readability; the full invariant can be found in the accompanying proof scripts.

Given these invariants, one can show that the return value `s` computed by `size` is consistent with the visibility set it constructs by picking up the memory tags from each node that it traverses. The loop invariant is more involved, as due to concurrent updates `size` could be traversing nodes that have been popped from the queue; see our CIVL proofs for more details.

Results Figure 3 provides a summary of our case studies. We separate the table into sections, one for each case study, and a common section at the top that contains the common theories of sets and sequences and our encoding of the heap. In each case study section, we separate the definitions of the atomic specification of the ADT (which can

```
// nextTags only contains singleton sets of push operations
(forall y: Ref ::
  (Btwn(next, start, y, null) && y != null && next[y] != null
    ==> nextTags[y] == Set(nextInvoc[y])
        && invoc_m(nextInvoc[y]) == Queue.push))

// nextTags of the last node is the empty set
&& nextTags[absRefs[Queue.stateTail(Queue.ofSeq(lin)) - 1]]
   == Set_empty()

// lin is made up of nextInvoc[y] for y in the queue
&& (forall n: Invoc :: invoc_m(n) == Queue.push
     ==> (Seq_elem(n, lin)
          <==> Btwn(next, start, nextRef[n], null)
               && nextRef[n] != null && next[nextRef[n]] != null))

// lin is ordered by order of nodes in queue
&& (forall n1, n2: Invoc ::
    (invoc_m(n1) == Queue.push && invoc_m(n2) == Queue.push
    && Seq_elem(n1, lin) && Seq_elem(n2, lin)
    ==> (Seq_ord(lin, n1, n2)
         <==> Btwn(next, nextRef[n1], nextRef[n1], nextRef[n2])
              && nextRef[n1] != nextRef[n2])))
```

Fig. 5. A snippet from the CIVL invariant for the queue.

be reused for other implementations) from the code and proof of the implementation we consider. For each resulting module, we list the number of lines of code, lines of proof, total lines, and CIVL's verification time in seconds. Experiments were conducted on an Intel Core i7-4470 3.4 GHz 8-core machine with 16GB RAM.

Our two case studies are representative of the weakly-consistent behaviors exhibited by all the Java concurrent objects studied in [13], both those using fixed-size arrays and those using dynamic memory. As CIVL does not direclty support dynamic memory and other Java language features, we were forced to make certain simplifications to the algorithms in our verification effort. However, the assumptions we make are orthogonal to the reasoning and proof of weak consistency of the monotonic methods. The underlying algorithm used by, and hence the proof argument for monotonicity of, hash map's `has` method is the same as that in the other monotonic hash map operations such as `elements`, `entrySet`, and `toString`. Similarly, the argument used for the queue's `size` can be adapted to other monotonic `ConcurrentLinkedQueue` and `LinkedTransferQueue` operations like `toArray` and `toString`. Thus, our proofs carry over to the full versions of the implementations as the key invariants linking the memory tags and visibility sets to the specification state are the same.

In addition, CIVL does not currently have any support for inferring the preconditions of each atomic block, which currently accounts for most of the lines of proof in our case studies. However, these problems have been studied and solved in other tools [30, 39], and in theory can be integrated with CIVL in order to simplify these kinds of proofs.

In conclusion, our case studies show that verifying weakly-consistent operations introduces little overhead compared to the proofs of the core absolute operations. The additional invariants needed to prove monotonicity were natural and easy to construct. We also see that our methodology brings weak-consistency proofs within the scope of what is provable by off-the-shelf automated concurrent program verifiers in reasonable time.

6 Related Work

Though *linearizability* [18] has reigned as the de-facto concurrent-object consistency criterion, several recent works proposed weaker criteria, including *quantitative relaxation* [17], *quiescent consistency* [10], and *local linearizability* [14]; these works effectively permit externally-visible interference among threads by altering objects' sequential specifications, each in their own way. Motivated by the diversity of these proposals, Sergey et al. [35] proposed the use of Hoare logic for describing a custom consistency specification for each concurrent object. Raad et al. [31] continued in this direction by proposing declarative consistency models for concurrent objects atop weak-memory platforms. One common feature between our paper and this line of work (see also [21, 9]) is encoding and reasoning directly about the concurrent history. The notion of *visibility relaxation* [13] originates from Burckhardt et al.'s axiomatic specifications [7], and leverages traditional sequential specifications by allowing certain operations to behave as if they are unaware of concurrently-executed linearization-order predecessors. The linearization (and visibility) actions of our simulation-proof methodology are unique to visibility-relaxation based weak-consistency, since they refer to a global linearization order linking executions with sequential specifications.

Typical methodologies for proving linearizability are based on reductions to safety verification [8, 5] and forward simulation [3, 37, 2], the latter generally requiring the annotation of per-operation *linearization points*, each typically associated with a single program statement in the given operation, e.g., a shared memory access. Extensions to this methodology include *cooperation* [38, 12, 41], i.e., allowing operations' linearization points to coincide with other operations' statements, and *prophecy* [33, 24], i.e., allowing operation' linearization points to depend on future events. Such extensions enable linearizability proofs of objects like the Herlihy-Wing Queue (HWQ). While prophecy [25], alternatively backward simulation [25], is generally more powerful than forward simulation alone, Bouajjani et al. [6] described a methodology based on forward simulation capable of proving seemingly future-dependent objects like HWQ by considering fixed linearization points only for value removal, and an additional kind of specification-simulated action, *commit points*, corresponding to operations' final shared-memory accesses. Our consideration of specification-simulated visibility actions follows this line of thinking, enabling the forward-simulation based proof of weakly-consistent concurrent objects.

7 Conclusion and Future Work

This work develops the first verification methodology for weakly-consistent operations using sequential specifications and forward simulation, thus reusing existing sequential ADT specifications and enabling simple reasoning, i.e., without prophecy [1] or backward simulation [25]. This paper demonstrates the application of our methodology to absolute and monotonic methods on sequentially-consistent memory, as these are the consistency levels demonstrated in actual Java implementations of which we are aware. Our formalization is general, and also applicable to the other visibility relaxations, e.g., the *peer* and *weak* visibilities [13], and weaker memory models, e.g., the Java memory model.

Extrapolating, we speculate that handling other visibilities amounts to adding annotations and auxiliary state which mirrors inter-operation communication. For example, while monotonic operations on shared-memory implementations observe mutating linearization-order predecessors – corresponding to a sequence of shared-memory updates – causal operations with message-passing based implementations would observe operations whose messages have (transitively) propagated. The corresponding annotations may require auxiliary state to track message propagation, similar in spirit to the `getModLin()` auxiliary state that tracks mutating linearization-order predecessors (§4). Since weak memory models essentially alter the mechanics of inter-operation communication, the corresponding visibility annotations and auxiliary state may similarly reflect this communication. Since this communication is partly captured by the denotations of memory commands (§3.2), these denotations would be modified, e.g., to include not one value and tag per memory location, but multiple. While variations are possible depending on the extent to which the proof of a given implementation relies on the details of the memory model, in the worst case the auxiliary state could capture an existing memory model (e.g., operational) semantics exactly.

As with systematic or automated linearizability-proof methodologies, our proof methodology is susceptible to two potential sources of incompleteness. First, as mentioned in Section 3, methodologies like ours based on forward simulation are only complete when specifications are *return-value deterministic*. However, data types are typically designed to be return-value deterministic and this source of incompleteness does not manifest in practice.

Second, methodologies like ours based on annotating program commands, e.g., with linearization points, are generally incomplete since the consistency mechanism employed by any given implementation may not admit characterization according to a given static annotation scheme; the Herlihy-Wing Queue, whose linearization points depend on the results of future actions, is a prototypical example [18]. Likewise, our systematic strategy for annotating implementations with *lin* and *vis* commands (§3) can fail to prove consistency of future-dependent operations. However, we have yet to observe any practical occurrence of such exotic objects; our strategy is sufficient for verifying the weakly-consistent algorithms implemented in the Java development kit. As a theoretical curiosity for future work, investigating the potential for complete annotation strategies would be interesting, e.g., for restricted classes of data types and/or implementations.

Finally, while CIVL's high-degree of automation facilitated rapid prototyping of our simulation proofs, its underlying foundation using Owicki-Gries style proof rules limits the potential for modular reasoning. In particular, while our weak-consistency proofs are thread-modular, our invariants and intermediate assertions necessarily talk about state shared among multiple threads. Since our simulation-based methodology and annotations are completely orthogonal to the underlying program logic, it would be interesting future work to apply our methodology using expressive logics like Rely-Guarantee, e.g. [19, 38], or variations of Concurrent Separation Logic, e.g. [28, 32, 34, 35, 4, 20]. It remains to be seen to what degree increased modularity may sacrifice automation in the application of our weak-consistency proof methodology.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grant No. 1816936, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

A Appendix: Proofs to Theorems and Lemmas

Lemma 1. *The abstract executions $E(W)$ of a specification W are consistent with W.*

Proof. Any complete, sequential, and absolute execution is consistent by definition, since the labeling of its linearization is taken from the sequential specification. Then, any happens-before weakening is consistent for exactly the same reason as its source execution, since its linearization and visibility projection are both identical. Finally, any visibility weakening is consistent by the condition of W-consistency in its definition. □

Lemma 2. *A weak-visibility specification and its transition system have identical histories.*

Proof. It follows almost immediately that the abstract executions of $[\![W]\!]_{\mathrm{s}}$ are identical to those of W, since $[\![W]\!]_{\mathrm{s}}$'s state effectively records the abstract execution of a given AETS execution, and only enables those returns that are consistent with W. Since histories are the projections of abstract executions, the corresponding history sets are also identical. □

Theorem 1. *A witnessing implementation I is consistent with a weak-visibility specification W if the transition system $[\![W]\!]_{\mathrm{s}}$ of W simulates I.*

Proof. This follows from standard arguments, given that the corresponding SLTSs include ε transitions to ensure that every move of one system can be matched by stuttering from the other: since both systems synchronize on the call, ret, hb, lin, and vis actions, the simulation guarantees that every abstract execution, and thus history, of I is matched by one of $[\![W]\!]_{\mathrm{s}}$. Then by Lemma 2, the histories of I are included in W. □

Theorem 3. *R_{chm} is a simulation relation from I_{chm} to $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$.*

Proof Sketch. We show that every step of the implementation, i.e., an atomic section or a program command, is simulated by $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$. Given $\langle q, e\rangle \in R_{\mathrm{chm}}$, we consider the different implementation steps which are possible in q.

The case of commands corresponding to procedure calls of `put` and `get` is trivial. Executing a procedure call in q leads to a new state q' which differs only by having a new active operation o. We have that $e \xrightarrow{\mathrm{call}(o,\_,\_)} e'$ and $\langle q', e'\rangle \in R_{\mathrm{chm}}$ where e' is obtained from e by adding o with an appropriate value of $inv(o)$ and an empty visibility.

The transition corresponding to the atomic section of `put` is labeled by a sequence of visibility actions (one for each linearized operation) followed by a linearization action. Let σ denote this sequence of actions. This transition leads to a state q' where the array `table` may have changed (unless writing the same value), and the history variable `lin` is extended with the `put` operation o executing this step. We define an abstract execution e' from e by changing lin to the new value of `lin`, and defining an absolute visibility for o. We have that $e \xrightarrow{\sigma} e'$ because e' is consistent with W_{m}. Also, $\langle q', e'\rangle \in R_{\mathrm{chm}}$ because the validity of (3), (4), and (5) follow directly from the definition

of e'. The atomic section of `get` can be handled in a similar way. The simulation of return actions of `get` operations is a direct consequence of point (6) which ensures consistency with S.

For `has`, we focus on the atomic sections containing `vis` commands and the linearization commands (the other internal steps are simulated by ϵ steps of $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$, and the simulation of the return step follows directly from (7) which justifies the consistency of the return value). The atomic section around the procedure call corresponds to a transition labeled by a sequence σ of visibility actions (one for each linearized modifier) and leads to a state q' with a new active `has` operation o (compared to q). We have that $e \xrightarrow{\sigma} e'$ because e' is consistent with W_{m}. Indeed, the visibility of o in e' is not constrained since o has not been linearized and the W_{m}-consistency of e' follows from the W_{m}-consistency of e. Also, $\langle q', e' \rangle \in R_{\mathrm{chm}}$ because $index(o) = -1$ and (7) is clearly valid. The atomic section around the read of `table[k]` is simulated by $[\![W_{\mathrm{m}}]\!]_{\mathrm{s}}$ in a similar way, noticing that (7) models precisely the effect of the visibility commands inside this atomic section. For the simulation of the linearization commands is important to notice that any active `has` operation in e has a visibility that contains all modifiers which returned before it was called and as explained above, this visibility is monotonic. □

References

[1] Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991)

[2] Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures for highly concurrent data structures. STTT **19**(5), 549–563 (2017)

[3] Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV. Lecture Notes in Computer Science, vol. 4590, pp. 477–490. Springer (2007)

[4] Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: Verification of parallel and concurrent software. In: IFM. Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017)

[5] Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. Inf. Comput. **261**(Part), 383–400 (2018)

[6] Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. In: CAV (2). Lecture Notes in Computer Science, vol. 10427, pp. 542–563. Springer (2017)

[7] Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: POPL. pp. 271–284. ACM (2014)

[8] Chakraborty, S., Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. Logical Methods in Computer Science **11**(1) (2015)

[9] Delbianco, G.A., Sergey, I., Nanevski, A., Banerjee, A.: Concurrent data structures linked in time. In: ECOOP. LIPIcs, vol. 74, pp. 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
[10] Derrick, J., Dongol, B., Schellhorn, G., Tofan, B., Travkin, O., Wehrheim, H.: Quiescent consistency: Defining and verifying relaxed linearizability. In: FM. Lecture Notes in Computer Science, vol. 8442, pp. 200–214. Springer (2014)
[11] Dongol, B., Jagadeesan, R., Riely, J., Armstrong, A.: On abstraction and compositionality for weak-memory linearisability. In: VMCAI. Lecture Notes in Computer Science, vol. 10747, pp. 183–204. Springer (2018)
[12] Dragoi, C., Gupta, A., Henzinger, T.A.: Automatic linearizability proofs of concurrent objects with cooperating updates. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 174–190. Springer (2013)
[13] Emmi, M., Enea, C.: Weak-consistency specification via visibility relaxation. PACMPL **3**(POPL), 60:1–60:28 (2019)
[14] Haas, A., Henzinger, T.A., Holzer, A., Kirsch, C.M., Lippautz, M., Payer, H., Sezgin, A., Sokolova, A., Veith, H.: Local linearizability for concurrent container-type data structures. In: CONCUR. LIPIcs, vol. 59, pp. 6:1–6:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
[15] Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. Logical Methods in Computer Science **6**(3) (2010)
[16] Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: CAV (2). Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015)
[17] Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: POPL. pp. 317–328. ACM (2013)
[18] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
[19] Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. pp. 321–332. North-Holland/IFIP (1983)
[20] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018)
[21] Khyzha, A., Dodds, M., Gotsman, A., Parkinson, M.J.: Proving linearizability using partial orders. In: ESOP. Lecture Notes in Computer Science, vol. 10201, pp. 639–667. Springer (2017)
[22] Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: ICALP (2). Lecture Notes in Computer Science, vol. 9135, pp. 311–323. Springer (2015)
[23] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
[24] Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. pp. 459–470. ACM (2013)
[25] Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. untimed systems. Inf. Comput. **121**(2), 214–233 (1995)

[26] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC. pp. 267–275. ACM (1996)
[27] Moskal, M., Lopuszanski, J., Kiniry, J.R.: E-matching for fun and profit. Electr. Notes Theor. Comput. Sci. **198**(2), 19–35 (2008)
[28] O'Hearn, P.W.: Resources, concurrency and local reasoning. In: CONCUR. Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004)
[29] Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Commun. ACM **19**(5), 279–285 (1976)
[30] Piskac, R., Wies, T., Zufferey, D.: Grasshopper - complete heap verification with mixed specifications. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 124–139. Springer (2014)
[31] Raad, A., Doko, M., Rozic, L., Lahav, O., Vafeiadis, V.: On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models. PACMPL **3**(POPL), 68:1–68:31 (2019)
[32] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
[33] Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 243–259. Springer (2012)
[34] Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI. pp. 77–87. ACM (2015)
[35] Sergey, I., Nanevski, A., Banerjee, A., Delbianco, G.A.: Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In: OOPSLA. pp. 92–110. ACM (2016)
[36] Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: CADE. Lecture Notes in Computer Science, vol. 3632, pp. 219–234. Springer (2005)
[37] Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI. Lecture Notes in Computer Science, vol. 5403, pp. 335–348. Springer (2009)
[38] Vafeiadis, V.: Automatically proving linearizability. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 450–464. Springer (2010)
[39] Vafeiadis, V.: Rgsep action inference. In: VMCAI. Lecture Notes in Computer Science, vol. 5944, pp. 345–361. Springer (2010)
[40] Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods. p. 561. North-Holland (1990)
[41] Zhu, H., Petri, G., Jagannathan, S.: Poling: SMT aided linearizability proofs. In: CAV (2). Lecture Notes in Computer Science, vol. 9207, pp. 3–19. Springer (2015)