Verifying Concurrent Multicopy Search Structures

NISARG PATEL, New York University, USA
SIDDHARTH KRISHNA, Microsoft Research, UK
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

Multicopy search structures such as log-structured merge (LSM) trees are optimized for high insert/update/delete (collectively known as upsert) performance. In such data structures, an upsert on key k, which adds (k, v) where v can be a value or a tombstone, is added to the root node even if k is already present in other nodes. Thus there may be multiple copies of k in the search structure. A search on k aims to return the value associated with the most recent upsert. We present a general framework for verifying linearizability of concurrent multicopy search structures that abstracts from the underlying representation of the data structure in memory, enabling proof-reuse across diverse implementations. Based on our framework, we propose template algorithms for (a) LSM structures forming arbitrary directed acyclic graphs and (b) differential file structures, and formally verify these templates in the concurrent separation logic Iris. We also instantiate the LSM template to obtain the first verified concurrent in-memory LSM tree implementation.

CCS Concepts: • **Theory of computation → Logic and verification**; **Separation logic**; **Shared memory algorithms**.

Additional Key Words and Phrases: template-based verification, concurrent data structures, log-structured merge trees, flow framework, separation logic

ACM Reference Format:
Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying Concurrent Multicopy Search Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 113 (October 2021), 32 pages. https://doi.org/10.1145/3485490

1 INTRODUCTION

Krishna et al. [2020a] demonstrated how to simplify the verification of concurrent search structure algorithms by abstracting implementations of diverse data structures such as B-trees, lists, and hash tables into templates that can be verified once and for all. The template algorithms considered in [Krishna et al. 2020a; Shasha and Goodman 1988] handle only search structures that perform all operations on keys *in-place*. That is, an operation on key k searches for the unique node containing k in the structure and then performs any necessary modifications on that node. Since every key occurs at most once in the data structure at any given moment, we refer to these structures as *single-copy (search) structures*.

Single-copy structures achieve high performance for reads. However, some applications, such as event logging, require high write performance, possibly at the cost of decreased read speed and increased memory overhead. This demand is met by data structures that store upserts (inserts, deletes or updates) to a key k *out-of-place* at a new node instead of overwriting a previous copy of

Authors' addresses: Nisarg Patel, New York University, USA, nisarg@nyu.edu; Siddharth Krishna, Microsoft Research, Cambridge, UK, siddharth@cs.nyu.edu; Dennis Shasha, New York University, USA, shasha@cims.nyu.edu; Thomas Wies, New York University, USA, wies@cs.nyu.edu.

2475-1421/2021/10-ART113
https://doi.org/10.1145/3485490

k that was already present in some other node. Performing out-of-place upserts can be done in constant time (e.g., always at the head of a list). A consequence of this design is that the same key k can now be present multiple times simultaneously in the data structure. Hence, we refer to these structures as *multicopy (search) structures*.

Examples of multicopy structures include the differential file structure [Severance and Lohman 1976], log-structured merge (LSM) tree [O'Neil et al. 1996], and the Bw-tree [Levandoski et al. 2013]. These concurrent data structures are widely used in practice, including in state-of-the-art database systems such as Apache Cassandra [Apache Software Foundation 2021] and Google LevelDB [Google 2021].

Like the verification method proposed by Krishna et al. [2020a], we aim to prove that the concurrent search structure of interest is linearizable [Herlihy and Tygar 1987], i.e., each of its operations appears to take effect atomically at a *linearization point* and behaves according to a sequential specification. For multicopy structures, the sequential specification is that of a (partial) mathematical map that maps a key to the last value that was upserted for that key. The framework proposed in [Krishna et al. 2020a; Shasha and Goodman 1988] does not extend to multicopy structures as it critically relies on the fact that every key is present in at most one node of the data structure at a time. Moreover, searches in multicopy structures exhibit dynamic non-local linearization points (i.e., the linearization point of a search is determined by and may be present during the execution of concurrently executing upserts). This introduces a technical challenge that is not addressed by this prior work. We discuss further related work in §9.

Contributions. This paper presents a framework for constructing linearizability proofs of concurrent multicopy structures with the goal of enabling proof reuse across data structures. Figure 1 provides an overview of our work. The paper starts by describing the basic intuition behind the correctness proof of any multicopy structure (§2). We then derive an abstract notion of multicopy structures similar to the abstract single-copy structures in the edgeset framework [Shasha and Goodman 1988] (§3). By introducing this intermediate abstraction level ("Template Level" in the figure) at which we can verify concurrent multicopy structure template algorithms, we aid proof reuse in two ways. First, the template algorithms abstract from the concrete representation of the data structure, allowing their proofs to be reused across diverse template instantiations. Second, the specification against which the templates are verified ("Search Recency") admits simpler linearizability proofs than the standard client-level specification of a search structure. The proof relating the client-level and template-level specification (§4) can be reused across all templates.

We demonstrate our framework by developing and verifying concurrent multicopy templates for (a) LSM structures and (b) differential file structures. The LSM template applies to existing LSM trees as well as to structures that form arbitrary directed acyclic graphs (DAGs) (§5, §6, and §7). The template and its proof support implementations based on different heap representations such as lists, arrays, and B-link trees. Verifying an instantiation of one of the two templates for a specific implementation involves only sequential reasoning about node-level operations.

We have mechanized both the proof relating client-level and template-level specifications as well as the verification of our template algorithms in the Coq-based interactive proof mode of the concurrent separation logic Iris [Jung et al. 2018; Krebbers et al. 2018, 2017]. Similar to [Krishna et al. 2020a], our formalization uses the flow framework [Krishna et al. 2018, 2020b] to enable local reasoning about inductive invariants of a general multicopy structure graph. In order to obtain a concrete multicopy structure, we have instantiated the node-level operations assumed by the LSM template for the LSM tree and verified their implementations in the automated separation logic verifier GRASShopper [Piskac et al. 2014] (§8). The result is the first formally-verified concurrent in-memory LSM tree implementation.

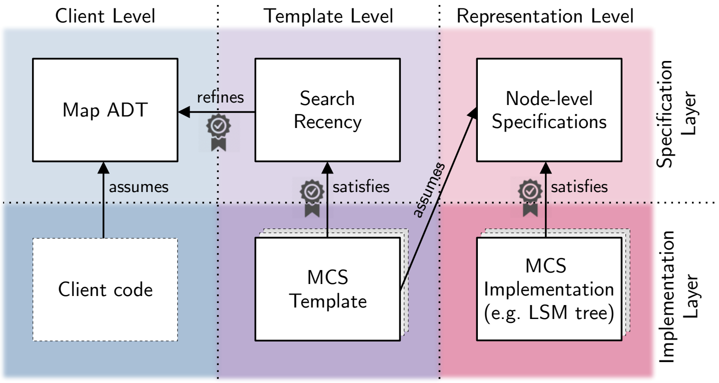

Fig. 1. The structure of our verification effort. MCS stands for multicopy structure.

2 MOTIVATION AND OVERVIEW

From a client's perspective, a multicopy structure implements a partial mathematical map $M\colon \mathsf{KS} \rightharpoonup \mathsf{V}$ of keys $k \in \mathsf{KS}$ to values $v \in \mathsf{V}$. We refer to M as the *logical contents* of the structure. The data structure supports insertions and deletions of key/value pairs on M and searches for the value $M(k)$ associated with a given key k.

The insert and delete operations are implemented by a single generic operation referred to as an *upsert*. The sequential specification of upsert is as follows. The operation takes a key-value pair (k, v) and updates M to $M[k \rightarrowtail v]$, associating k with the given value v. To delete a key k from the structure, one upserts the pair $(k, \square)$ where $\square$ is a dedicated *tombstone* value used to indicate that k has been deleted. The sequential specification of a search for a key k is then as expected: it returns $M(k)$ if M is defined for k and $\square$ otherwise.

Multicopy structures are commonly used in scenarios where the nodes representing the data structure's logical contents M are spread over multiple media such as memory, solid-state drives, and hard disk drives. Each node therefore contains its own data structure that is designed for the particular characteristics of the underlying medium, typically an unsorted array at the root to allow upserts to perform fast appends and a classical single-copy search structure (e.g., a hash structure or arrays with bloom filters) for non-root nodes. The non-root nodes are typically read-only, so concurrency at the node level is not an issue. In this paper, we consider the multicopy data structure as a graph of nodes. We study template algorithms on that graph.

2.1 A Library Analogy to Multicopy Search Structures

To train your intuition about multicopy structures, consider a library of books in which new editions of the same book arrive over time. Thus the first edition of book k can enter and later the second edition, then the third and so on. A patron of this library who enters the library at time t and is looking for book k should find an edition that is either current at time t or one that arrives in the library after t. We call this normative property *search recency*.

Now suppose the library is organized as a sequence of rooms. All new books are put in the first room (near the entrance). When a new edition v of a book arrives in the first room, any previous editions of that book in that room are thrown out. When the first room becomes full, the books in that room are moved to the second room. If a previous edition of some book is already in the second room, that previous edition is thrown out. When the second room becomes full, its books are moved to the third room using the same throwing out rule, and so on. This procedure maintains

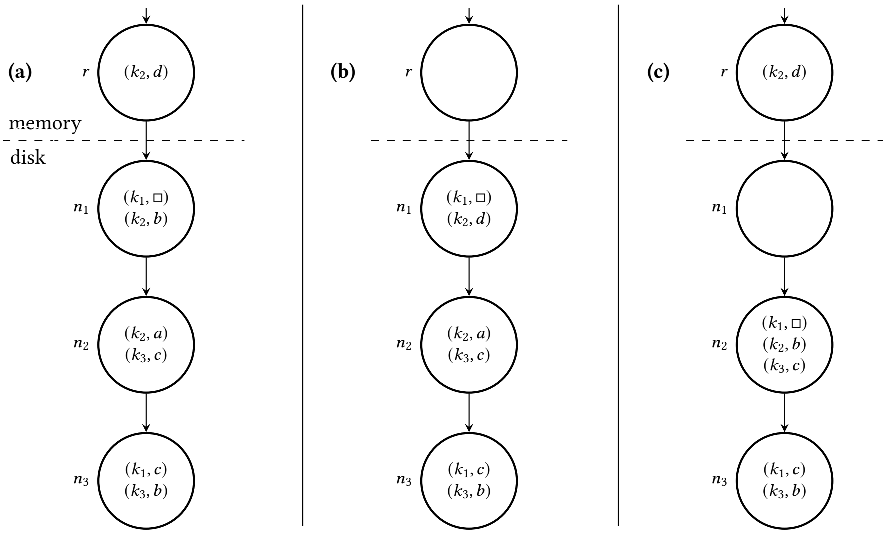

Fig. 2. **(a)** High-level structure of an LSM tree. **(b)** LSM tree obtained from (a) after flushing node r to disk. **(c)** LSM tree obtained (a) after compacting nodes n_1 and n_2.

the time-ordering invariant that the editions of the same book are ordered from most recent (at or nearer to the first room) to least recent (farther away from the first room) in the sequence of rooms.

A patron's search for k starting at time t begins in the first room. If the search finds any edition of k in that room, the patron takes a photocopy of that edition. If not, the search proceeds to the second room and so on.

Now suppose that the latest edition at time t is edition v and there is a previous edition v'. Because of the time-ordering invariant and the fact that the search begins at the first room, the search will encounter v before it encounters v'. The search may also encounter an even newer edition of k, but will never encounter an older one before returning. That establishes the search recency property.

Any concurrent execution of inserts and searches is equivalent to a serial execution in which (i) each insert is placed in its relative order of entering the root node with respect to other inserts and (iia) a search s is placed after the insert whose edition s copies if that insert occurred after s began or (iib) a search s is placed at the point when s began, if the edition that s copies was inserted before s began (or if s returns no edition at all).

Because the searches satisfy the search recency property, the concurrent execution is *linearizable* [Herlihy and Wing 1990], which is our ultimate correctness goal.

Note that the analogy as written has treated only inserts and searches. However, updates and deletions can be implemented as inserts: an update to book k can be implemented as the insertion of a new edition; a delete of book k can be implemented as the insertion of an edition whose value is a "tombstone" which is an indication that book k has been deleted.

2.2 Log-Structured Merge Trees

A prominent example of a multicopy structure is the LSM tree, which closely corresponds to the library analogy described above. The data structure consists of a root node r stored in memory (the first room in the library), and a linked list of nodes $n_1, n_2, \ldots, n_l$ stored on disk (the remaining rooms). Figure 2 (a) shows an example.

The LSM tree operations essentially behave as outlined in the library analogy. The upsert operation takes place at the root node r. A search for a key k traverses the list starting from the root node and retrieves the value associated with the first copy of k that is encountered. If the retrieved value is $\square$ or if no entry for k has been found after traversing the entire list, then the search determines that k is not present in the data structure. Otherwise, it returns the retrieved value. For instance, a search for key k_1 on the LSM tree depicted in Figure 2 (a) would determine that this key is not present since the retrieved value is $\square$ from node n_1. Similarly, k_4 is not present since there is no entry for this key. On the other hand, a search for k_2 would return d and a search for k_3 would return c.

To prevent the root node from growing too large, the LSM tree performs *flushing*. As the name suggests, the flushing operation flushes the data from the root node to the disk by moving its contents to the first disk node. Figure 2 (b) shows the LSM tree obtained from Figure 2 (a) after flushing the contents of r to the disk node n_1.

Similar to flushing, a *compaction* operation moves data from full nodes on disk to their successor. In case there is no successor, then a new node is created at the end of the structure. During the merge, if a key is present in both nodes, then the most recent (closer-to-the-root) copy is kept, while older copies are discarded. Figure 2 (c) shows the LSM tree obtained from Figure 2 (a) after compacting nodes n_1 and n_2. Here, the copy of k_2 in n_2 has been discarded. In practice, the length of the data structure is bounded by letting the size of newly created nodes grow exponentially.

The net effect of all these operations is that the data structure satisfies the time-ordering invariant and searches achieve search recency.

The LSM tree can be tuned by implementing workload- and hardware-specific data structures at the node level. In addition, research has been directed towards optimizing the layout of nodes and developing different strategies for the maintenance operations used to reorganize these data structures. This has resulted in a variety of implementations today (e.g. [Dayan and Idreos 2018; Luo and Carey 2020; Raju et al. 2017; Thonangi and Yang 2017; Wu et al. 2015]). Despite the differences between these implementations, they generally follow the same high-level algorithms for the core search structure operations.

We construct template algorithms for concurrent multicopy structures from the high-level descriptions of their operations and then prove the correctness of these operations. Notably our LSM DAG template generalizes the LSM tree so that the outer data structure can be a DAG rather than just a list. A number of existing LSM structures are based on trees (e.g. [Sears and Ramakrishnan 2012; Wu et al. 2015]). Practical implementations of tree-based concurrent search structures often have additional pointer structures layered on top of the tree that make them DAGs. For instance, many implementations use the *link technique* to increase performance. Here, when a maintenance operation relocates a key k from one node to another, it adds a pointer linking the two nodes, which ensures that k remains reachable via the old search path. A concurrent thread searching for k that arrives at the old node can then follow the link, avoiding a restart of the search from the root. Our verified templates can be instantiated to lock-based implementations of this technique.

3 MULTICOPY SEARCH STRUCTURE FRAMEWORK

We build our formal framework of multicopy structures on the concurrent separation logic Iris [Jung et al. 2018]. A detailed introduction to Iris is beyond the scope of this paper. We therefore introduce only the relevant features of the logic as we use them.

3.1 Multicopy Search Structures

We abstract away from the data organization within the nodes, and treat the data structure as consisting of nodes in a mathematical directed acyclic graph.

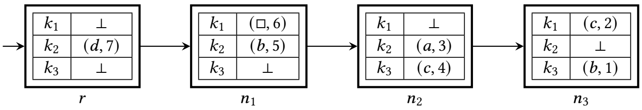

Fig. 3. Abstract multicopy data structure graph for the LSM tree in Figure 2 (a).

Since copies of a single key k can be present in different nodes simultaneously, we need a mechanism to differentiate between these copies. To that end, we augment each entry (k, v) stored in a node with the unique timestamp t identifying the point in time when (k, v) was upserted: $(k, (v, t))$. The timestamp plays the role of the book edition in the library analogy from the last section. For example in Figure 3, (k_3, c) was upserted after (k_2, a), which was upserted after (k_3, b). To generate these timestamps, we use a single global clock, which we initialize to 1. Note that the timestamp associated with an upserted value is auxiliary, or *ghost*, data that we use in our proofs to track the temporal ordering of the copies present in the structure at any point. Implementations do not need to explicitly store this timestamp information.

Formally, let KS be the set of all keys and V a set of values with a dedicated tombstone value $\square \in \mathsf{V}$. A multicopy (search) structure is a directed acyclic graph $G = (N, E)$ with nodes N and edges $E \subseteq N \times N$. We assume that there is a dedicated *root node* $r \in N$ which uniquely identifies the structure. Each node n of the graph is labeled by its contents $C_n \colon \mathsf{KS} \rightharpoonup \mathsf{V} \times \mathbb{N}$, which is a partial map from keys to pairs of values and timestamps. For a node n and its contents C_n, we say $(k, (v, t))$ is in the contents of n if $C_n(k) = (v, t)$. We denote the absence of an entry for a key k in n by $C_n(k) = \bot$ and let $\mathsf{dom}(C_n) := \{k \mid C_n(k) \neq \bot\}$. We further write $\mathsf{val}(C_n) \colon \mathsf{KS} \rightharpoonup \mathsf{V}$ for the partial function that strips off the timestamp information from the contents of a node, $\mathsf{val}(C_n) := \lambda k.\ (\exists v. C_n(k) = (v, \_)\ ?\ v : \bot)$.

For each edge $(n, n') \in E$ in the graph, the *edgeset* $\mathsf{es}(n, n')$ is the set of keys k for which an operation arriving at a node n would traverse (n, n') if $k \notin \mathsf{dom}(C_n)$. We require that the edgesets of all outgoing edges of a node n are pairwise disjoint. Figure 3 shows a potential abstract multicopy structure graph consistent with the LSM tree depicted in Figure 2 (a). Here, all edges have edgeset KS.

3.2 Client-Level Specification

Our goal is to prove the linearizability of concurrent multicopy structure templates with respect to their desired sequential client-level specification. As discussed earlier, the sequential specification is that of a map ADT, i.e., the *logical contents* of the data structure is a mathematical map from keys to values, $M \colon \mathsf{KS} \to \mathsf{V}$. The map M associates every key k with the most recently upserted value v for k, respectively, $\square$ if k has not yet been upserted:

$$M(k) := \begin{cases} v & \text{if } \exists n\, t.\ C_n(k) = (v, t) \wedge t = \max\{t' \mid \exists n'\, v'.\ C_{n'}(k) = (v', t')\} \\ \square & \text{otherwise} \end{cases}$$

We call $M(k)$ the *logical value* of key k.

Linearizability of a data structure is defined in terms of the concurrent execution histories of the data structure's operations [Herlihy and Tygar 1987; Herlihy and Wing 1990]. Hoare logics like Iris emphasize proof decomposition, which means, in particular, that they strive to reason only about a single data structure operation at a time. It is therefore difficult to specify linearizability directly in such logics. Instead, we specify the intended behavior of each data structure operation in terms of an *atomic triple* [da Rocha Pinto et al. 2014; Frumin et al. 2018; Jacobs and Piessens 2011; Jung et al. 2020, 2015]. Atomic triples can be thought of as the concurrent counterparts of sequential Hoare triples. They formalize the intuition that a linearizable operation appears to take effect atomically at a single point in time, the operation's *linearization point.*

More precisely, an atomic triple $\left\langle \vec{x}.\, P \right\rangle\ e\ \left\langle v.\, Q \right\rangle$ is made up of a precondition P, which may refer to the variables $\vec{x}$, a postcondition Q, which relates the variables $\vec{x}$ and the return value v, and a program e. The triple states that e may assume that for each of its atomic steps up to its linearization point, the shared state satisfies P for possibly different values of $\vec{x}$ in each step. At the linearization point, e then changes the shared state to one that satisfies Q in one atomic step. Afterwards, e no longer access resources in P or Q. Intuitively, concurrently executing threads may interfere with e by modifying the shared state but they are required to maintain P as an invariant.

Now suppose that $\overline{\mathsf{MCS}}(r, M)$ is a *representation predicate* that provides the client view of a multicopy structure with root r, abstracting its shared state by the logical contents M. We then require that the search and upsert methods respect the following client-level atomic specifications:

$$\left\langle M.\, \overline{\mathsf{MCS}}(r, M) \right\rangle \ \mathtt{upsert}\ r\ k\ v\ \left\langle \overline{\mathsf{MCS}}(r, M[k \rightarrowtail v]) \right\rangle \tag{1}$$

$$\left\langle M.\, \overline{\mathsf{MCS}}(r, M) \right\rangle \ \mathtt{search}\ r\ k\ \left\langle v.\, \overline{\mathsf{MCS}}(r, M) * M(k) = v \right\rangle \tag{2}$$

The specification of upsert updates the logical value of k to v. Thus upsert performs the "insert" of the library analogy. The search specification states that search returns the logical value $M(k) = v$ of its query key k.

3.3 Template-Level Specification: Search Recency

The verification of multicopy structures requires reasoning about the dynamic non-local linearization points of search, which are determined by the concurrently executing upserts. We want to avoid having to do this reasoning each time we verify a new template for a multicopy structure implementation. Our strategy is to provide an alternative template-level specification that uses a more detailed abstraction of the computation history rather than just the logical contents. This alternative specification will then have fixed local linearization points, simplifying the verification.

We say that search satisfies *search recency* if each concurrent invocation search $r\,k$ either returns the logical value associated with k at the point when the search started, or any other copy of k that was upserted between the search's start time and the search's end time.

We will show that if searches satisfy search recency and upserts take effect in a single atomic step that changes the logical contents M according to (1), then the multicopy structure is linearizable.

We start by defining the *upsert history* $H \subseteq \mathsf{KS} \times (\mathsf{V} \times \mathbb{N})$ of a multicopy data structure as the set of all copies $(k, (v, t))$ that have been upserted thus far. In particular, we require that any multicopy structure will maintain the following predicates concerning H and the global clock t:

$$\mathsf{HInit}(H) \coloneqq \forall k.\ (k, (\square, 0)) \in H$$

$$\mathsf{HUnique}(H) \coloneqq \forall k\, t'\, v_1\, v_2.\ (k, (v_1, t')) \in H \land (k, (v_2, t')) \in H \Rightarrow v_1 = v_2$$

$$\mathsf{HClock}(t, H) \coloneqq \forall (k, (\_, t')) \in H.\ t' < t$$

The predicate $\mathsf{HUnique}(H)$ ensures that we can lift the total order $t_1 \leqslant t_2$ on timestamps to a total order $(v_1, t_1) \leqslant (v_2, t_2)$ on the pairs of values and timestamps occurring in H. The lifted order simply ignores the value component. Together with $\mathsf{HInit}(H)$, this ensures that the following function is well-defined:

$$\bar{H} \coloneqq \lambda k.\ \max\left\{(v, t) \mid (k, (v, t)) \in H\right\}.$$

The latest copy of a key will always be contained in some node n of the data structure. If the data structure implementation maintains the additional invariant, $H \supseteq \bigcup_{n \in N} C_n$, then this guarantees that $\bar{H}$ is consistent with the logical contents M, i.e., for all keys k, $\bar{H}(k) = (M(k), \_)$. Finally, the predicate $\mathsf{HClock}(t, H)$ guarantees that $\mathsf{HUnique}(H)$ is preserved when a new entry $(k, (v, t))$ is added to H for the current value of the global clock t.

Assume that, similar to $\overline{\mathsf{MCS}}(r, M)$, we are given a template-level representation predicate $\mathsf{MCS}(r, t, H)$ that abstracts the state of a multicopy structure by its upsert history H and the current value t of the global clock. The desired template-level specification of `upsert` in terms of the new abstraction is simply:

$$\big\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\big\rangle\ \mathtt{upsert}\ r\ k\ v\ \big\langle\, \mathsf{MCS}(r, t+1, H \cup (k, (v, t))) \,\big\rangle \tag{3}$$

It states that `upsert` advances the value of the global clock from t to $t + 1$ and adds a new copy $(k, (v, t))$ to the upsert history H.

The postcondition of `search` needs to express two properties. First, we must necessarily have $(k, (v, t')) \in H$, where v is the value returned by `search`, t' is v's associated timestamp, and H is the value of the upsert history at the linearization point. Moreover, let H_0 be the value of the upsert history at the start of the search and define $(v_0, t_0) \coloneqq \bar{H}_0(k)$. Then either v is the logical value of k at that point (i.e. $v = v_0$) or t' is the timestamp of an upsert for k that happened after the search started, i.e., $t_0 < t'$. This is equivalent to demanding that for all t_0' and v' such that $(k, (v', t_0')) \in H_0$, the returned timestamp t' satisfies $t_0' \leqslant t'$. We define the auxiliary abstract predicate $\mathsf{SR}(k, v, t)$ to mean that $(k, (v, t)) \in H$ for the value H of the upsert history at the time point when the predicate is evaluated.[1] Using this predicate, the template-level specification of `search` is then expressed as follows:

$$\begin{array}{l} \forall v_0'\ t_0'.\ \mathsf{SR}(k, v_0', t_0') \mathrel{-\!\!*} \\ \quad \big\langle\, t\, H.\ \mathsf{MCS}(r, t, H) \,\big\rangle\ \mathtt{search}\ r\ k\ \big\langle\, v.\ \exists t'.\ \mathsf{MCS}(r, t, H) \ *\ t_0' \leqslant t' \ *\ (k, (v, t')) \in H \,\big\rangle \end{array} \tag{4}$$

Here, we use the *magic wand* connective $\mathrel{-\!\!*}$ to express that the auxiliary local precondition $\mathsf{SR}(k, v_0', t_0')$ must be satisfied at the time point when `search` is invoked.

In the next section, we deal with the complexity of non-local dynamic linearization points of searches once and for all by proving that any multicopy structure that satisfies the template-level specification also satisfies the desired client-level specification. To prove the correctness of a given concurrent multicopy structure, it then suffices to show that `upsert` satisfies its corresponding template-level specification (3) and `search` satisfies (4). When proving the validity of the template-level atomic triple of `search` for a particular implementation (or template), one can now always *commit* the atomic triple (i.e., declare a linearization point) at the point when the return value v of the search is determined, i.e., when $(k, (v, t')) \in H$ is established. This linearization point is now independent of concurrently executing `upsert`s.

4 RELATING THE CLIENT-LEVEL AND TEMPLATE-LEVEL SPECIFICATIONS

We next prove that any concurrent execution of `upsert` and `search` operations that satisfy the template-level specifications (3) and (4) can be linearized to an equivalent sequential execution that satisfies the client-level specifications. Intuitively, this can be done by letting the `upsert`s in the equivalent sequential execution occur in the same order as their atomic commit points in the concurrent execution, and by letting each `search` $r\ k$ occur at the earliest time after the timestamp t' associated with the returned value v of k. That is, if $v = v_0$ (recall that $(v_0, t_0) = \bar{H}(k)$ where H is the upsert history at the start of the search on k), then the search occurs right after it was invoked in the concurrent execution. Otherwise we must have $t' > t_0$ and the search occurs after the upsert at time t'. The fact that such an upsert must exist follows from the template-level specifications.

[1] In §4.2 we will express $\mathsf{SR}(k, v, t)$ using appropriate Iris ghost state that keeps track of the upsert history.

The intuitive proof argument above relies on explicit reasoning about execution histories. Instead, we aim for a thread-modular proof that reasons about individual `search`es and `upsert`s in isolation, so that we can mechanize the proof in a Hoare logic like Iris. The proof we present below takes inspiration from that of the RDCSS data structure by Jung et al. [2020].

4.1 Challenges and Proof Outline

Iris prophecies [Jung et al. 2020], based on the idea first introduced by Abadi and Lamport [1988], allow a thread to predict what will happen in the future. In particular, one can use prophecies to predict future events in order to reason about non-fixed linearization points [Vafeiadis 2008; Zhang et al. 2012]. In our case, a thread executing `search` can use a prophecy to predict, at the beginning of the search, the value v that it will eventually return. In a thread-modular correctness proof, one can then decide on how to linearize the operation based on the predicted value.

The linearization point of a `search` operation occurs when an instruction of a concurrent `upsert` is executed. One can view this as a form of *helping* [Liang and Feng 2013]: when an `upsert` operation commits and adds $(k, (v, t'))$ to the upsert history H, it also commits all the (unboundedly many) concurrently executing `search` operations for k that will return v. We encode this *helping protocol* in the predicate $\overline{\mathsf{MCS}}(r, M)$ that captures the shared (ghost) state of the data structure, by taking advantage of Iris's support for higher-order ghost state.

At a high level, the proof then works as follows. We augment `search` with auxiliary ghost code that creates and resolves the relevant prophecies. We do this by defining the wrapper function $\overline{\texttt{search}}$ given in Figure 4. The right side shows the specifications of the two functions related to manipulating (one-shot) prophecies in Iris. The function `NewProph` returns a fresh prophecy p that predicts the value v_p. This fact is captured by the resource $\mathsf{Proph}(p, v_p)$ in the postcondition of the Hoare triple specifying `NewProph`. The resource $\mathsf{Proph}(p, v_p)$ can be owned by a thread as well as transferred between threads via shared resources such as the representation predicate $\mathsf{MCS}(r, t, H)$ (as is usual in concurrent separation logics). The resource is also exclusive, meaning it cannot be duplicated.

The function $\overline{\texttt{search}}$ uses `NewProph` to create two prophecies, which it binds to *tid* and p. The prophecy p predicts the value v that will eventually be returned by `search`. The value *tid* predicted by the second prophecy will be used later as a unique identifier of the thread performing the search when we encode the helping protocol, taking advantage of the fact that each prophecy returned by `NewProph` is fresh. Freshness of prophecies also ensures that each prophecy can be resolved only once, which is done using `Resolve` p `to` v. This operation consumes the resource $\mathsf{Proph}(p, v_p)$ and yields the proposition $v_p = v$. It is used on line 5 of $\overline{\texttt{search}}$ to express that the value predicted by p is indeed the value v returned by `search`.

If v is equal to the current logical value v_0 of k at the start of $\overline{\texttt{search}}$, then the proof commits the client-level atomic triple right away. If instead $v_0 \neq v$, then the proof *registers* the thread's client-level atomic triple in the shared predicate $\overline{\mathsf{MCS}}(r, M)$. The registered atomic triple serves as an obligation to commit the atomic triple. This obligation will be discharged by the `upsert` operation adding $(k, (v, t'))$ to H. The proof of $\overline{\texttt{search}}$ then uses the template-level specification of `search` to conclude that it can collect the committed triple from the shared predicate after `search` has returned.

Relating the high-level and low-level specification of `upsert` is straightforward. However, the proof of `upsert` also needs to do its part of the helping protocol by scanning over all the searches that are currently registered in the shared predicate $\overline{\mathsf{MCS}}(r, M)$ and committing those that return the copy of k added by the `upsert`.

In the remainder of this section we explain this helping proof in more detail.

```
let $\overline{\texttt{search}}$ r k =
  let tid = NewProph in
  let p = NewProph in
  let v = search r k in
  Resolve p to v; v
```

ONE-SHOT-PROPHECY-CREATION

$$\{\mathsf{True}\}\ \mathtt{NewProph}\ \{p.\ \exists v_p.\ \mathsf{Proph}(p, v_p)\}$$

ONE-SHOT-PROPHECY-RESOLUTION

$$\{\mathsf{Proph}(p, v_p)\}\ \mathtt{Resolve}\ p\ \mathtt{to}\ v\ \{v_p = v\}$$

Fig. 4. Wrapper augmenting search with prophecy-related ghost code, whose specification is on the right.

4.2 Keeping Track of the Upsert History

Our thread-modular proof exploits the observation that the upsert history H only increases over time. Thus, assertions such as $(k, (v, t)) \in H$, as used in our specification of search recency, are stable under interference. This style of reasoning follows the classic idea of establishing lower bounds on monotonically evolving state (see e.g. [Fahndrich and Leino 2003; Jensen and Birkedal 2012; Jones 1983]). We formalize this in Iris using user-defined ghost state.

Iris expresses ownership of ghost state by the proposition $\boxed{a}^{\gamma}$ which asserts ownership of a piece a of the ghost location γ. It is the ghost analogue of the points-to predicate $x \mapsto v$ in separation logic, except that $\boxed{a}^{\gamma}$ asserts only that γ contains a value *one of whose parts* is a. This means ghost state can be split and combined according to the rules of the *camera*, the algebraic structure from which the values (like a) are drawn. Cameras generalize partial commutative monoids, which are commonly used to give semantics to separation logics. A camera comes equipped with a set M and a binary composition operation $(\cdot)\colon M \times M \to M$ that form a commutative monoid. The composition operation gives meaning to the separating conjunction of predicates that express fragmental ownership of ghost state at a ghost location γ via the rule: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$. A simple example of a camera is $\textsc{Set}(X)$, where X is some set. Here, $M = 2^X$ and $(\cdot)$ is set union. Another example is the *heap camera* of standard separation logic, which consists of mappings from heap locations to values that can be composed by disjoint set union.

Iris also provides generic "functors" for constructing new cameras from existing ones. One example that we will be using in our proofs is the *authoritative* camera $\textsc{Auth}(M)$, which can be constructed from any other camera M. It is used to model situations where threads share an authoritative element a of M via a representation predicate and individual threads own fragments b of a. We denote an authoritative element by $\bullet a$ and a fragment by $\circ b$. The composition $\bullet a \cdot \circ b$ expresses ownership of the authoritative element a and, in addition, $\exists c.\ a = b \cdot c$.

For instance, we use the *authoritative set* camera $\textsc{Auth}(\textsc{Set}(\mathsf{KS} \times (\mathsf{V} \times \mathbb{N})))$ to keep track of the upsert history H at a ghost location γ_s. The proposition $\boxed{\bullet H}^{\gamma_s}$ states that H is the current authoritative version of the upsert history. This ghost resource is kept in the representation predicate $\mathsf{MCS}(r, t, H)$, which is shared among all threads operating on the data structure. The camera can also express lower bounds $H' \subseteq H$ on the authoritative set H using propositions of the form $\boxed{\circ H'}^{\gamma_s}$. That is, the proposition $\boxed{\bullet H}^{\gamma_s} * \boxed{\circ H'}^{\gamma_s}$ asserts ownership of the current upsert history H and, in addition, $H' \subseteq H$. We can then define the predicate $\mathsf{SR}(k, v, t)$, which expresses that $\bar{H}(k) = (v, t)$ was true at some point in the past, as $\mathsf{SR}(k, v, t) \coloneqq \boxed{\circ \{(k, (v, t))\}}^{\gamma_s}$.

Iris allows *frame-preserving updates* of ghost state, denoted by the *view shift* connective $\Rrightarrow$. For instance, the following rules capture some frame-preserving updates of authoritative sets:

AUTH-SET-UPD

$$\frac{H \subseteq H'}{\boxed{\bullet H}^{\gamma} \Rrightarrow \boxed{\bullet H'}^{\gamma}}$$

AUTH-SET-SNAP

$$\boxed{\bullet H}^{\gamma} \Rrightarrow \boxed{\bullet H}^{\gamma} * \boxed{\circ H}^{\gamma}$$

AUTH-SET-FRAG

$$\boxed{\circ H}^{\gamma} * \boxed{\circ H'}^{\gamma} \Longleftrightarrow \boxed{\circ (H \cup H')}^{\gamma}$$

$$
\begin{aligned}
\overline{\mathsf{MCS}}(r, M) &:= \exists\, t\, H.\ \mathsf{MCS}(r, t, H) \ast \forall k.\ (M(k), \_) = \bar{H}(k) \\
\mathsf{MCS}(r, t, H) &:= \boxed{\bullet\ H}^{\gamma_s} \ast \mathsf{HInit}(H) \ast \mathsf{HUnique}(H) \ast \mathsf{HClock}(t, H) \\
&\qquad \ast \mathsf{Inv}_{tpl}(t, H) \ast \mathsf{Prot}_{help}(H) \\
\mathsf{Prot}_{help}(H) &:= \exists R.\ \boxed{\bullet\ R}^{\gamma_r} \ast \mathop{\bigast}_{tid \in R} \exists\, k\, v_p\, t_0\, \Phi\, \mathsf{Tok}.\ \mathsf{Proph}(tid, \_) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \ast \mathsf{State}(H, k, v_p, t_0, \Phi, \mathsf{Tok}) \\
\mathsf{State}(H, k, v_p, t_0, \Phi, \mathsf{Tok}) &:= \mathsf{Pending}(H, k, v_p, t_0, \Phi) \lor \mathsf{Done}(H, k, v_p, t_0, \Phi, \mathsf{Tok}) \\
\mathsf{Pending}(H, k, v_p, t_0, \Phi) &:= \mathsf{AU}(\Phi) \ast (\forall t.\ (k, (v_p, t)) \in H \Rightarrow t < t_0) \\
\mathsf{Done}(H, k, v_p, t_0, \Phi, \mathsf{Tok}) &:= (\Phi(v_p)\ \lor\ \mathsf{Tok}) \ast (\exists t.\ (k, (v_p, t)) \in H \land t \geqslant t_0)
\end{aligned}
$$

Fig. 5. Definition of client-level representation predicate and invariants of helping protocol.

The rule AUTH-SET-UPD is the only way to update the authoritative element, because, intuitively, it must maintain the validity of all lower bounds. The authoritative set camera thus implicitly enforces the invariant that the upsert history can only increase. We use this rule to update the authoritative version of the upsert history at the linearization point of `upsert`.

The rule AUTH-SET-SNAP allows us to take a "snapshot" of the current authoritative set. We use this rule together with the rule AUTH-SET-FRAG at the call to `search` in $\overline{\texttt{search}}$ to establish the thread-local precondition $\mathsf{SR}(k, v_0', t_0')$ of the specification (4) from the shared resource $\boxed{\bullet H}^{\gamma_s}$. To this end, we choose $(v_0', t_0') := \bar{H}(k)$, which gives us $H = H \cup \{(k, (v_0', t_0'))\}$ and thus $\boxed{\circ \{(k, (v_0', t_0'))\}}^{\gamma_s}$.

4.3 The Helping Protocol

Before we discuss the details of our encoding of the helping protocol in terms of Iris ghost state, let us recall the basic structure of a proof of an atomic triple $\langle x.\ P \rangle\ e\ \langle v.\ Q \rangle$. The proof proceeds by proving a standard Hoare triple of the form $\forall \Phi.\ \{\mathsf{AU}_{x.P,Q}(\Phi)\}\ e\ \{v.\ \Phi(v)\}$. Here, $\mathsf{AU}_{x.P,Q}(\Phi)$ is the *atomic update token*, which gives us the *right* to use the resources in the precondition P when executing atomic instructions up to the linearization point. The token also records our *obligation* to preserve P up to the linearization point, where P must be transformed to Q in one atomic step. This step consumes the update token. The universally quantified proposition Φ can be thought of as the precondition for the continuation of the client of the atomic triple. At the linearization point, when the atomic update token is consumed, the corresponding proof rule produces $\Phi(v)$ as a receipt that the obligation has been fulfilled. This receipt is necessary to complete the proof of the Hoare triple. Figure 5 shows a simplified definition of $\overline{\mathsf{MCS}}(r, M)$ and the invariant that encodes the helping protocol.[2] The definitions are implicitly parameterized by a proposition $\mathsf{Inv}_{tpl}(r, t, H)$, which abstracts from the resources needed for proving that a specific multicopy structure template satisfies the template-level specifications. In particular, this invariant will store the resources needed to represent the node-level contents C_n for each node $n \in N$. It also ties the C_n to H, capturing the invariant $H \supseteq \bigcup_{n \in N} C_n$.

[2] For presentation purposes, the proof outline presented here abstracts from some technical details of the actual proof done in Iris. For a more detailed presentation of our Iris development, we refer the interested reader to [Patel et al. 2021, Appendix A].

The predicate $\overline{\mathsf{MCS}}(r, M)$ contains the predicate $\mathsf{MCS}(r, t, H)$, used in the template-level atomic triples, and defines M in terms of $\bar{H}$. The predicate $\mathsf{MCS}(r, t, H)$ owns all (ghost) resources associated with the data structure. In particular, this predicate stores the ghost resource $\boxed{\bullet H}^{\gamma_s}$, holding the authoritative version of the current upsert history, the abstract template-level invariant $\mathsf{Inv}_{tpl}(r, t, H)$, and the helping protocol predicate $\mathsf{Prot}_{help}(H)$, described below. $\mathsf{MCS}(r, t, H)$ also states the three invariants $\mathsf{HInit}(H)$, $\mathsf{HUnique}(t, H)$, and $\mathsf{HClock}(H)$ discussed earlier, which are needed to prove the atomic triple of $\overline{\texttt{search}}$.

The helping protocol predicate Prot_{help} contains a *registry* $\boxed{\bullet\ R}^{\gamma_r}$ of $\overline{\texttt{search}}$ thread IDs that require helping from `upsert` threads. For each thread ID *tid* in the registry, the shared state contains $\mathsf{Proph}(tid, \_)$ along with the state of *tid*, which is either Pending or Done. Pending captures an uncommitted $\overline{\texttt{search}}$, and Done describes the operation after it has been committed. Note that we omit the annotation of the pre and postcondition from $\mathsf{AU}(\Phi)$ as it always refers to the specification of `search` in this proof.

The proof outline for $\overline{\texttt{search}}$ is shown in Figure 12. After creating the two prophecies *tid* and *p*, the proof case-splits on whether the thread requires helping or not (line 9). We only consider the helping case (i.e., $(v_0, t_0) = \bar{H}_0(k) \wedge v_0 \neq v_p$), where H_0 is the initial upsert history and v_p the prophesied return value). Here, the thread registers itself with the helping protocol by replacing R with $R \cup \{tid\}$ using rule AUTH-SET-UPD (line 14). To do this, it first establishes $\mathsf{Pending}(H_0, k, v_p, t_0, \Phi)$ by transferring its obligation to linearize to the shared state, captured by the update token $\mathsf{AU}(\Phi)$. The condition $\forall t. (k, (v_p, t)) \in H_0 \Rightarrow t < t_0$ follows from $v_0 \neq v_p$, the definition of $\bar{H}_0$, and the invariant $\mathsf{HUnique}(H_0)$. The thread also a creates a fresh non-duplicable token Tok that it will later trade in for the receipt $\Phi(v_p)$.

Let us briefly switch to the role played by the `upsert` that updates the logical value of k to v_p at some time $t \geqslant t_0$. When this `upsert` reaches its linearization point, our proof uses rule AUTH-SET-UPD to update the upsert history from H to $H \cup \{(k, (v_p, t))\}$, as required by the postcondition of (3), and also increments the global clock from t to $t + 1$. We must then show that $\mathsf{MCS}(r, t + 1, H \cup \{(k, (v_p, t))\})$ holds after these ghost updates, which requires us to prove $\mathsf{Prot}_{help}(H \cup \{(k, (v_p, t))\})$ assuming $\mathsf{Prot}_{help}(H)$ was true before the update. In particular, any $\overline{\texttt{search}}$ thread that was in a Pending state $\mathsf{State}(H, k, v_p, t_0, \Phi, \mathsf{Tok})$ and thus waiting to be helped by this `upsert` needs to be committed. It can do this because the postcondition of these triples are satisfied after H has been updated. The proof then transfers the receipts $\Phi(v_p)$ back to the shared representation predicate, yielding new states $\mathsf{Done}(H \cup \{(k, (v_p, t))\}, k, v_p, t_0, \Phi, \mathsf{Tok})$ for each of these threads.

Coming back to the proof of the $\overline{\texttt{search}}$ that needed helping, after the call to `search` on line 16, we know from the postcondition of (4) that we must have $(k, (v, t')) \in H$ for some t' such that $t' \geqslant t_0$ where H is the new upsert history at this point. Moreover, after resolving the prophecy on line 18 we know $v_p = v$ and therefore $(k, (v_p, t')) \in H$. From the invariant, we can then conclude that the thread must be in a Done state. Since the thread owns the unique token Tok, it trades it in to obtain $\Phi(v)$, which lets it complete the proof of its atomic triple specification (4).

5 THE LSM DAG TEMPLATE

This section presents a general template for multicopy structures that generalizes the LSM (log-structured merge) tree discussed in §2.2. We prove linearizability of the template by verifying that all operations satisfy the template-level atomic triples (§3.3). The template and the proof parameterize over the implementation of the single-copy data structures used at the node-level. Instantiating the template for a specific implementation involves only sequential reasoning about the implementation-specific node-level operations.

```
$\langle M.\ \overline{\mathsf{MCS}}(r, M) \rangle$
let $\overline{\texttt{search}}$ $r$ $k$ =
  $\{\mathsf{AU}(\Phi)\}$
  let $tid$ = NewProph in
  let $p$ = NewProph in
  $\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * \mathsf{Proph}(p, v_p)\}$
  $\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * \mathsf{Proph}(p, v_p) * \mathsf{MCS}(r, t, H_0)\}$
  $\left\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * \mathsf{Proph}(p, v_p) * \boxed{\circ\ H_0}^{\gamma_s} * (v_0, t_0) = \bar{H}_0(k) * \mathsf{SR}(k, v_0, t_0)\right\}$
  (* Case analysis on $v_p = v_0$, $v_p \neq v_0$: only showing $v_p \neq v_0$ *)
  $\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * \mathsf{Proph}(p, v_p) * (v_0, t_0) = \bar{H}_0(k) * \mathsf{SR}(k, v_0, t_0) * v_p \neq v_0 * \ldots\}$
    $\{\ldots * \mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * (\forall t.\ (k, (v_p, t)) \in H_0 \Rightarrow t < t_0)\}$
    $\{\ldots * \mathsf{Proph}(tid, \_) * \mathsf{Pending}(H_0, k, vp, t_0, \Phi) * \mathsf{Tok}\}$
    $\left\{\ldots * \boxed{\bullet\ R}^{\gamma_r} * tid \notin R * \mathsf{State}(H_0, k, v_p, t_0, \Phi, \mathsf{Tok}) * \mathsf{Tok}\right\}$
  (* Ghost update: $\boxed{\bullet\ R}^{\gamma_r} \Rrightarrow \boxed{\bullet\ R \cup \{tid\}}^{\gamma_r}$ *)
  $\left\{\mathsf{Proph}(p, v_p) * \mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * \mathsf{SR}(k, v_0, t_0) * v_p \neq v_0 * \mathsf{MCS}(r, t, H)\right\}$
  let $v$ = search $r$ $k$ in
  $\left\{\mathsf{Proph}(p, v_p) * \mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * v_p \neq v_0 * \mathsf{MCS}(r, t, H) * t_0 \leqslant t' * (k, (v, t')) \in H\right\}$
  Resolve $p$ to $v$;
  $\left\{\mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * v \neq v_0 * \mathsf{MCS}(r, t, H) * t_0 \leqslant t' * (k, (v, t')) \in H\right\}$
  $\left\{\cdots * \mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * v \neq v_0 * t_0 \leqslant t' * (k, (v, t')) \in H * \mathsf{State}(H, k, v, t_0, \Phi, \mathsf{Tok})\right\}$
  $\left\{\cdots * \mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * t_0 \leqslant t' * (k, (v, t')) \in H * \mathsf{Done}(H, k, v, t_0, \Phi, \mathsf{Tok})\right\}$
  $\left\{\cdots * \mathsf{Tok} * \boxed{\circ\ \{tid\}}^{\gamma_r} * t_0 \leqslant t' * (k, (v, t')) \in H * (\Phi(v) \vee \mathsf{Tok})\right\}$
  $\left\{\cdots * \Phi(v) * \boxed{\circ\ \{tid\}}^{\gamma_r} * t_0 \leqslant t' * (k, (v, t')) \in H * (\Phi(v) \vee \mathsf{Tok})\right\}$
  $\left\{\cdots * \Phi(v) * \boxed{\circ\ \{tid\}}^{\gamma_r} * \mathsf{Done}(H, k, v, t_0, \Phi, \mathsf{Tok})\right\}$
  $\{\Phi(v) * \mathsf{MCS}(r, t, H)\}$
  $\left\{\overline{\mathsf{MCS}}(r, M) * M(k) = v\right\}$
  $v$
  $\langle v.\ \overline{\mathsf{MCS}}(r, M) * M(k) = v \rangle$
```

Fig. 6. Outline for the proof of the client-level specification for $\overline{\texttt{search}}$.

We split the template into two parts. The first part is a template for `search` and `upsert` that works on general multicopy structures, i.e., arbitrary DAGs with locally disjoint edgesets. The second part (discussed in §7) is a template for a maintenance operation that generalizes the compaction mechanism found in existing list-based LSM tree implementations to tree-like multicopy structures.

Figure 7 shows the code of the template for the core multicopy operations. The operations `search` and `upsert` closely follow the high-level description of these operations on the LSM tree (§2.2). The operations are defined in terms of implementation-specific helper functions `findNext`, `addContents`, and `inContents`.

The `search` operation calls the recursive function `traverse` on the root node. `traverse` $r\ n\ k$ first locks the node n and uses the helper function `inContents` $r\ n\ k$ to check if a copy of key k

```
let rec traverse r n k =
  lockNode n;
  match inContents r n k with
  | Some v -> unlockNode n; v
  | None ->
    match findNext r n k with
    | Some n' ->
      unlockNode n;
      traverse r n' k
    | None -> unlockNode n; □

let search r k = traverse r r k
```

```
let rec upsert r k v =
  lockNode r;
  let res = addContents r k v in
  if res then
    unlockNode r
  else begin
    unlockNode r;
    upsert r k v
  end
```

Fig. 7. The general template for multicopy operations search and upsert. The template can be instantiated by providing implementations of helper functions inContents, findNext, and addContents. inContents $r\,n\,k$ returns Some v if $(v, t') = C_n(k)$ for some t', and None otherwise. findNext $r\,n\,k$ returns Some n' if n' is the unique node such that $k \in \mathrm{es}(n, n')$, and None otherwise. addContents $r\,k\,v$ updates the contents of r by setting the value associated with key k to v. The return value of addContents is a Boolean which indicates whether the insertion was successful (e.g., if r is full, insertion may fail leaving r's contents unchanged).

is contained in n. If a copy of k is found, then its associated value v is returned after unlocking n. Otherwise, traverse uses the helper function findNext to determine the unique successor n' of the given node n and query key k (i.e., the node n' satisfying $k \in \mathrm{es}(n, n')$). If such a successor n' exists, traverse recurses on n'. Otherwise, traverse concludes that there is no copy of k in the data structure and returns □. Note that this algorithm uses fine-grained concurrency, as the thread executing the search holds at most one lock at any point (and no locks at the points when traverse is called recursively).

The upsert $r\ k\ v$ operation locks the root node and adds a new copy of the key k with value v to the contents of the root node using addContents. addContents $r\ k\ v$ adds the pair (k, v) to the root node when it succeeds. upsert terminates by unlocking the root node. The addContents function may however fail if the root node is full. In this case upsert calls itself recursively[3].

6 VERIFYING THE TEMPLATE

We next discuss the correctness proof of the template operations. We will focus on the high-level proof ideas and key invariants and defer the detailed proof outline and encoding of the invariants in Iris to [Patel et al. 2021, Appendix A].

6.1 High-Level Proof Outline

Proof of search. We start with the proof of search. Recall that search recency is the affirmation that if t_0 is the logical timestamp of k at the point when search $r\,k$ is invoked, then the operation returns v such that $(k, (v, t')) \in H$ and $t' \geqslant t_0$. Since the value v of k retrieved by search comes from some node in the structure, we must examine the relationship between the upsert history H of the data structure and the physical contents C_n of the nodes n visited as the search progresses. We do this by identifying the main invariants needed for proving search recency for arbitrary multicopy structures.

[3] For simplicity of presentation, we assume that a separate maintenance thread flushes the root if it is full to ensure that upserts eventually make progress.

We refer to the *spatial ordering* of the copies $(k, (v, t))$ stored in a multicopy structure as the ordering in which those copies are reached when traversing the data structure graph starting from the root node. Our first observation is that the spatial ordering is consistent with the temporal ordering in which the copies have been upserted. We referred to this property as the time-ordering invariant in our library analogy in §2.1: the farther from the root a search is, the older the copies it finds are. Therefore, if a `search` $r\ k$ traverses the data structure without interference from other threads and returns the first copy of k that it finds, then it is guaranteed to return the logical value of k at the start of the search.

We formalize this observation in terms of the *contents-in-reach* of a node. The contents-in-reach of a node n is the partial function $C_{ir}(n) \colon \mathsf{KS} \rightharpoonup \mathsf{V} \times \mathbb{N}$ defined recursively over the graph of the multicopy structure as follows:

$$C_{ir}(n)(k) \coloneqq \begin{cases} C_n(k) & \text{if } k \in \mathsf{dom}(C_n) \\ C_{ir}(n')(k) & \text{else if } \exists n'.\, k \in \mathsf{es}(n, n') \\ \bot & \text{otherwise} \end{cases} \tag{5}$$

Note that $C_{ir}(n)$ is well-defined because the graph is acyclic and the edgesets labeling the outgoing edges of every node n are disjoint. We further define $\mathsf{ts}(C_{ir}(n)(k)) = t$ if $C_{ir}(n)(k) = (\_, t)$ and $\mathsf{ts}(C_{ir}(n)(k)) = 0$ if $k \notin \mathsf{dom}(C_{ir}(n))$.

For example, in the multicopy structure depicted in Figure 3, we have $C_{ir}(r) = \{k_1 \rightarrowtail (\square, 6), k_2 \rightarrowtail (d, 7), k_3 \rightarrowtail (c, 4)\}$ and $C_{ir}(n_3) = C_{n_3}$.

The observation that interference-free searches will find the current logical timestamp of their query key is then captured by the following invariant:

Invariant 1 The logical contents of the multicopy structure is the contents-in-reach of its root node: $\bar{H} = C_{ir}(r)$.

In order to account for concurrent threads interfering with the search, we prove the condition $t_0 \leq t'$ for the timestamp t' associated with the value returned by the search. Intuitively, this is true because the contents-in-reach of a node n can be affected only by upserts or maintenance operations, both of which only increase the timestamps associated with every key of any given node: upserts insert new copies into the root node and maintenance operations move recent copies down in the structure, possibly replacing older copies. This observation is formally captured by the following invariant:

Invariant 2 The contents-in-reach of every node only increases. That is, for every node n and key k, if $\mathsf{ts}(C_{ir}(n)(k)) = t$ at some point in time and $\mathsf{ts}(C_{ir}(n)(k)) = t'$ at any later point in time, then $t \leq t'$.

Finally, in order to prove the condition $(k, (v, t')) \in H$ of search recency, we need one additional property:

Invariant 3 All copies present in the multicopy structure have been upserted at some point in the past. That is, for all nodes n, $C_n \subseteq H$.

Now let us consider an execution of `search` on a operation key k. In addition to the above three general invariants, we need an inductive invariant for the traversal performed by the search: we require as a precondition for `traverse` $r\ n\ k$ that $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$ where t_0 is the timestamp of the logical value v_0 of k at the point when `search` was invoked. To see that this property holds initially for the call to `traverse` $r\ r\ k$ in `search`, let $\bar{H}_0$ be the logical contents at the time point when `search` was invoked. The precondition $\mathsf{SR}(k, v_0, t_0)$ implies $\mathsf{ts}(\bar{H}_0(k)) \geqslant t_0$, which, combined with Invariant 1 implies that we must have had $\mathsf{ts}(C_{ir}(r)(k)) \geqslant t_0$ at this point. Since $\mathsf{ts}(C_{ir}(r)(k))$ only increases over time because of Invariant 2, we can conclude that $\mathsf{ts}(C_{ir}(r)(k)) \geqslant t_0$ when traverse is called. We next show that the traversal invariant is maintained by traverse and is sufficient to prove search recency.

Consider a call to traverse $r\,n\,k$ such that $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$ holds initially. We must show that the call returns v such that $(k,(v,t')) \in H$ and $t' \geqslant t_0$ for some t'. We know that the call to inContents on line 3 returns either Some v such that $(v,t') = C_n(k)$ or None if $C_n(k) = \bot$. Let us first consider the case where inContents returns Some v. In this case, traverse returns v on line 4. By definition of $C_{ir}(n)$ we have $C_{ir}(n)(k) = C_n(k)$. Hence, we have $\mathsf{ts}(C_{ir}(n)(k)) = t'$ and the precondition $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$, together with Invariant 2, implies $t' \geqslant t_0$. Moreover, Invariant 3 guarantees $(k,(v,t')) \in H$.

Now consider the case where inContents returns None. Here, $k \notin \mathrm{dom}\, C_n(k)$, indicating that no copy has been found for k in n. In this case, traverse calls findNext to obtain the successor node of n and k. In the case where the successor n' exists (line 7), we know that $k \in \mathsf{es}(n,n')$ must hold. Hence, by definition of contents-in-reach we must have $C_{ir}(n)(k) = C_{ir}(n')(k)$. From $\mathsf{ts}C_{ir}(n)(k) \geqslant t_0$ and Invariant 2, we can then conclude $\mathsf{ts}(C_{ir}(n')(k)) \geqslant t_0$, i.e. that the precondition for the recursive call to traverse on line 9 is satisfied and search recency follows by induction.

On the other hand, if n does not have any next node, then traverse returns □ (line 10), indicating that k has not yet been upserted at all so far (i.e., has never appeared in the structure). In this case, by definition of contents-in-reach we must have $C_{ir}(n)(k) = \bot$. Invariant 2 then guarantees $\mathsf{ts}(C_{ir} n k) = 0 = t_0$. The invariant $\mathsf{HInit}(H)$ on the upsert history then gives us $(k,(\square,0)) \in H$. Hence, search recency holds in this case for $t' = 0$.

Proof of upsert. In order to prove the logically atomic specification (11) of upsert, we must identify an atomic step where the clock t is incremented and the upsert history H is updated. Intuitively, this atomic step is when the lock on the root node is released (line 17 in Figure 7) after addContents succeeds. Note that in this case addContents changes the contents of the root node from C_r to $C_r' = C_r[k \rightarrowtail (v,t)]$. Hence, in the proof we need to update the ghost state for the upsert history from H to $H' = H \cup \{(k,(v,t))\}$, reflecting that a new copy of k has been upserted. It then remains to show that the three key high-level invariants of multicopy structures identified above are preserved by these updates.

First, observe that Invariant 3, which states $\forall n.\ C_n \subseteq H$, is trivially maintained: only C_r is affected by the upsert and the new copy $(k,(v,t))$ is included in H'. Similarly, we can easily show that Invariant 2 is maintained: $C_{ir}(n)$ remains the same for all nodes $n \neq r$ and for the root node it increases, provided Invariant 1 is also maintained.

Thus, the interesting case is Invariant 1. Proving that this invariant is maintained amounts to showing that $\bar{H}'(k) = (v,t)$. This step critically relies on the following additional observation:

Invariant 4 All timestamps in H are smaller than the current time of the global clock t.

This invariant implies that $\bar{H}'(k) = \max(\bar{H}(k),(v,t)) = (v,t)$, which proves the desired property. We note that Invariant 4 is maintained because the global clock is incremented when H is updated to H', and, as we describe below, while r is locked.

In the remainder of this section, we discuss the key technical issue when formalizing the above proof in a separation logic like Iris.

6.2 Iris Invariant

The Iris proof must capture the key invariants identified in the proof outline given above in terms of appropriate ghost state constructions. We start by addressing the key technical issue that arises when formalizing the above proof in a separation logic like Iris: contents-in-reach is a recursive function defined over an arbitrary DAG of unbounded size. This makes it difficult to obtain a simple local proof that involves reasoning only about the bounded number of modified nodes in the graph.

The recursive and global nature of contents-in-reach mean that modifying even a single edge in the graph can potentially change the contents-in-reach of an unbounded number of nodes (for example, deleting an edge (n_1, n_2) can change $C_{ir}(n)$ for all n that can reach n_1). A straightforward attempt to prove that a template algorithm preserves Invariant 2 would thus need to reason about the entire graph after every modification (for example, by performing an explicit induction over the full graph). We solve this challenge using the flow framework [Krishna et al. 2020b].

Encoding Contents-in-Reach using Flows. The flow framework enables separation-logic-style reasoning about recursive functions on graphs. Certain restrictions apply. The function must be of the form $fl\colon N \to M$ where N is the set of nodes of the graph and $(M, +, 0)$ is a commutative cancellative monoid, called the *flow domain.* Further, fl must satisfy the *flow equation*:

$$\forall n \in N.\ fl(n) = in(n) + \sum_{n' \in N} e(n', n)(fl(n')) \tag{FlowEqn}$$

Intuitively, this equation states that fl can be computed by assigning every node an initial value according to the *inflow function* $in\colon N \to M$ and then propagating these values along the edges of the graph using the *edge function* $e\colon N \times N \to M \to M$ to reach a fixpoint. At each node n, the values propagated from predecessor nodes n' are aggregated using the monoid operation $+$. A function fl that satisfies the flow equation is called a *flow* and a graph equipped with a flow is a *flow graph.* The flow framework then enables us to reason compositionally about invariants of flow graphs expressed as node-local conditions that depend on a node's flow.

If we can define the contents-in-reach in terms of a flow, then we can use the notion of a *flow interface* to prove locally that an update to the graph does not change the flow of any nodes outside the modified region. The flow interface of a region consists of its outflow and inflow, maps that intuitively capture the contribution of this region to the flow of the rest of the world and the contribution of the outside world to this region's flow, respectively. If the interface of a modified region is preserved, then the framework guarantees that the flow of the rest of the graph is unchanged. Thus, our proofs need to prove only that Invariant 2 is preserved for a bounded set of affected nodes.

Technically, this kind of reasoning is enabled by the separation algebra structure of flow graphs (in particular the definition of flow graph composition), which extends the composition of partial graphs in standard separation logic so that the frame rule also preserves flow values of nodes in the frame. Instead of performing an explicit induction over the entire graph structure to prove that contents-in-reach values continue to satisfy desired invariants, the necessary induction is hidden away inside the definition of flow graph composition (for more details see [Krishna et al. 2020b]). Note that since `search` does not modify the multicopy structure, it trivially maintains the flow interface of the nodes it operates on, and hence any flow-based invariants.

Equation (5) defines contents-in-reach in a bottom-up fashion, starting from the leaves of the multicopy structure graph. That is, the computation proceeds *backwards* with respect to the direction of the graph's edges. This makes a direct encoding of contents-in-reach in terms of a flow difficult because the flow equation describes computations that proceed in the forward direction.

We side-step this problem by tracking auxiliary ghost information in the data structure invariant for each node n in the form of a function $Q_n\colon \mathsf{KS} \rightharpoonup \mathsf{V} \times \mathbb{N}$. If these ghost values satisfy

$$Q_n = \lambda k. \begin{cases} C_{ir}(n')(k) & \text{if } \exists n'.\ k \in \mathsf{es}(n, n') \\ \bot & \text{otherwise} \end{cases} \tag{6}$$

and we additionally define

$$B_n \coloneqq \lambda k.\ (k \in \mathsf{dom}(C_n(k))\ ?\ C_n(k) : Q_n(k))$$

then $C_{ir}(n) = B_n$. The idea is that each node stores Q_n so that node-local invariants can use it to talk about $C_{ir}(n)$. We then use a flow to propagate the purported values Q_n forward in the graph to ensure that they indeed satisfy (6). Note that while an `upsert` or maintenance operations on n may change B_n, it preserves Q_n. That is, operations do not affect the contents-in-reach of downstream nodes, allowing local reasoning about the modification of the contents of n.

In what follows, let us fix a multicopy structure over nodes N and some valuations of the partial functions Q_n. The flow domain M for our encoding of contents-in-reach consists of multisets of key/value-timestamp pairs $M \coloneqq \mathsf{KS} \times (\mathsf{V} \times \mathbb{N}) \to \mathbb{N}$ with multiset union as the monoid operation. The edge function induced by the multicopy structure is defined as follows:

$$e(n, n')(\_) \coloneqq \chi(\{(k, Q_n(k)) \mid k \in \mathsf{es}(n, n') \wedge k \in \mathsf{dom}(Q_n)\}) \tag{7}$$

Here, χ takes a set to its corresponding multiset. Additionally, we let the function *in* map every node to the empty multiset. With the definitions of e and *in* in place, there exists a unique flow *fl* that satisfies (FlowEqn). Now, if every node n in the resulting flow graph satisfies the following two predicates

$$\phi_1(n) \coloneqq \forall k.\ Q_n(k) = \bot \vee (\exists n'.\, k \in \mathsf{es}(n, n')) \tag{8}$$

$$\phi_2(n) \coloneqq \forall k\, p.\ fl(n)(k, p) > 0 \Rightarrow B_n(k) = p \tag{9}$$

then $B_n = C_{ir}(n)$. Note that the predicates ϕ_1 and ϕ_2 depend only on n's own flow and its local ghost state (i.e., Q_n, C_n and the outgoing edgesets $\mathsf{es}(n, \_)$).

Encoding the Invariants in Iris. We can now define the template-specific invariant $\mathsf{Inv}_{tpl}(r, t, H)$ for the LSM DAG template, which is assumed by the representation predicate $\mathsf{MCS}(\mathsf{Inv}_{tpl}, \mathsf{Prot})(r, t, H)$ defined in Figure 5. We denote this invariant by Inv_{LSM} and it is defined as follows:

$$\mathsf{Inv}_{LSM}(r, t, H) \coloneqq \exists N.\ \mathsf{G}(r, t, H, N) * \mathop{\Huge *}_{n \in N} \begin{array}{l} \exists b_n\, C_n\, Q_n.\ \ \mathsf{L}(b_n, n, \mathsf{N_L}(r, n, C_n, Q_n)) \\ \qquad\qquad * \ \ \mathsf{N_S}(r, n, C_n, Q_n, H) \end{array}$$

The existentially quantified variable N denotes the set of nodes of the multicopy structure. The invariant itself consists of two parts. The predicate $\mathsf{G}(r, t, H, N)$ states certain invariants about its parameters and contains *global* ghost resources storing the values t and N. The second part is an iterated separating conjunction stating ownership of the node-local resources associated with every node $n \in N$.

The resources associated with each node n are split between two predicates. The predicate $\mathsf{N_S}(r, n, C_n, Q_n, H)$ holds those resources associated with n that can be accessed by any thread operating on the data structure regardless of whether n is locked or not. In particular, it contains the two predicates $\phi_1(n)$ and $\phi_2(n)$ needed for our encoding of contents-in-reach. The second predicate $\mathsf{N_L}(r, n, C_n, Q_n)$ contains all resources that are accessible only to a thread that currently holds the lock on n. Ownership of node-local ghost state such as Q_n is shared between the two predicates. This ensures that a thread may update the values of these resources only when it holds n's lock. Moreover, every thread can assume that the constraints imposed on these values by $\mathsf{N_S}$ are true, even at times when the thread does not hold the lock.

$\mathsf{N_L}(r, n, C_n, Q_n)$ includes $\mathsf{node}(r, n, \mathsf{es}(n, \cdot), \mathsf{val}(C_n))$, which is a predicate that encapsulates all resources specific to the implementation of the node-specific data structure abstracted by node n. In particular, this predicate owns the resources associated with the physical representation of the data structure and ties them to the abstract ghost state representing the high-level multicopy structure: the node's physical contents $\mathsf{val}(C_n)$ (i.e., C_n without timestamps) and the edgesets of its outgoing edges $\mathsf{es}(n, \cdot)$. Our template proof is parametric in the definition of node and depends only on the following two assumptions that each implementation used to instantiate the template

```
$\langle\, b\, R.\ \mathsf{L}(b, n, R) \,\rangle$ lockNode $n$ $\langle\, \mathsf{L}(\mathit{true}, n, R) * R \,\rangle$
$\langle\, R.\ \mathsf{L}(\mathit{true}, n, R) * R \,\rangle$ unlockNode $n$ $\langle\, \mathsf{L}(\mathit{false}, n, R) \,\rangle$
$\{\mathsf{node}(r, n, es, V_n)\}$ inContents $n$ $k$ $\{x.\ \mathsf{node}(r, n, es, V_n) * x = (k \in \mathsf{dom}(V_n)\ ?\ \mathsf{Some}(V_n(k)) : \mathsf{None})\}$
$\{\mathsf{node}(r, n, es, V_n)\}$ findNext $n$ $k$ $\{x.\ \mathsf{node}(r, n, es, V_n) * x = (\exists n'.\ k \in es(n')\ ?\ \mathsf{Some}(n') : \mathsf{None})\}$
$\{\mathsf{node}(r, r, es, V_r)\}$ addContents $r$ $k$ $v$ $\{b.\ \mathsf{node}(r, r, es, V_r') * V_r' = (b\ ?\ V_r[k \rightarrowtail v] : V_r)\}$
```

Fig. 8. Specifications of helper functions used by search and upsert.

must satisfy. First, we require that node is not duplicable:

$$\mathsf{node}(r, n, es, V_n) * \mathsf{node}(r', n, es', V_n') \vdash \mathsf{False}$$

Moreover, node must guarantee disjoint edgesets:

$$\mathsf{node}(r, n, es, V_n) \vdash \forall n_1\, n_2.\ n_1 = n_2 \lor es(n_1) \cap es(n_2) = \emptyset$$

The predicate $\mathsf{L}(b, n, R_n)$ captures the abstract state of n's lock and is used to specify the protocol providing exclusive access to the resource R_n protected by the lock via the helper functions lockNode and unlockNode. The Boolean b indicates whether the lock is (un)locked. The specifications of the helper functions used by search and upsert, given in terms of the predicates $\mathsf{L}(b, n, R_n)$ and $\mathsf{node}(r, n, es, V_n)$ are shown in Figure 8. We discuss further details in [Patel et al. 2021, Appendix A].

7 MULTICOPY MAINTENANCE OPERATIONS

We next show that we can extend our multicopy structure template in §5 with a generic maintenance operation without substantially increasing the proof complexity. The basic idea of our proofs here is that for every timestamped copy of key k, denoted as the pair $(k, (v, t))$, every maintenance operation either does not change the distance of $(k, (v, t))$ to the root or increases it while preserving an edgeset-guided path to $(k, (v, t))$. Using these two facts, we can prove that all the structure invariants are also preserved.

7.1 Maintenance template

For the maintenance template, we consider a generalization of the compaction operation found in LSM tree implementations such as LevelDB [Google 2021] and Apache Cassandra [Apache Software Foundation 2021; Jonathan Ellis 2011]. While those implementations work on lists for the high-level multicopy structure, our maintenance template supports arbitrary tree-like multicopy structures. The code is shown in Figure 9. The template uses the helper function atCapacity r n to test whether the size of n (i.e., the number of non-$\bot$ entries in n's contents) exceeds an implementation-specific threshold. If not, then the operation simply terminates. In case n is at capacity, the function chooseNext is used to determine the node to which the contents of n can be merged. If the contents of n can be merged to successor m of n, then chooseNext returns Some m. In case no such successor exists, then it returns None. If chooseNext returns Some m, then the contents of n are merged to m. By merge, we mean that some copies of keys are transferred from n to m, possibly replacing older copies in m. The merge is performed by the helper function mergeContents. It must ensure that all keys k merged from C_n to C_m satisfy $k \in \mathsf{es}(n, m)$.

On the other hand, if chooseNext returns None, then a new node is allocated using the function allocNode. The new node is then added to the data structure using the helper function insertNode. Here, the new edgeset $\mathsf{es}(n, m)$ must be disjoint from all edgesets for the other successors m' of n. Afterwards, the contents of n are merged to m as before. Note that the maintenance template never removes nodes from the structure. In practice, the depth of the structure is bounded by letting the

```
let rec compact r n =
  lockNode n;
  if atCapacity r n then begin
    match chooseNext r n with
    | Some m ->
      lockNode m;
      mergeContents r n m;
      unlockNode n;
      unlockNode m;
      compact r m
    | None ->
      let m = allocNode () in
      insertNode r n m;
      mergeContents r n m;
      unlockNode n;
      unlockNode m;
      compact r m
  end
  else
    unlock n
```

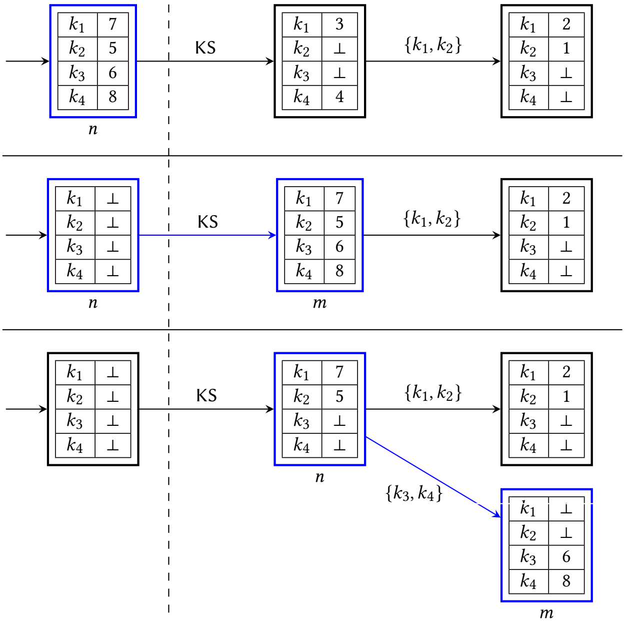
k_1 7, k_2 5, k_3 6, k_4 8
n
KS
k_1 3, k_2 ⊥, k_3 ⊥, k_4 4
$\{k_1, k_2\}$
k_1 2, k_2 1, k_3 ⊥, k_4 ⊥
k_1 ⊥, k_2 ⊥, k_3 ⊥, k_4 ⊥
n
KS
k_1 7, k_2 5, k_3 6, k_4 8
m
$\{k_1, k_2\}$
k_1 2, k_2 1, k_3 ⊥, k_4 ⊥
k_1 ⊥, k_2 ⊥, k_3 ⊥, k_4 ⊥
KS
k_1 7, k_2 5, k_3 ⊥, k_4 ⊥
n
$\{k_1, k_2\}$
k_1 2, k_2 1, k_3 ⊥, k_4 ⊥
$\{k_3, k_4\}$
k_1 ⊥, k_2 ⊥, k_3 6, k_4 8
m

Fig. 9. Maintenance template for tree-like multicopy structures. The template can be instantiated by providing implementations of helper functions `atCapacity`, `chooseNext`, `mergeContents`, `allocNode`, and `insertNode`. `atCapacity` $r\ n$ returns a Boolean value indicating whether node n has reached its capacity. The helper function `chooseNext` $r\ n$ returns Some m if there exists a successor m of n in the data structure into which n should be compacted, and None in case n cannot be compacted into any of its successors. `mergeContents` $r\ n\ m$ (partially) merges the contents of n into m. Finally, `allocNode` is used to allocate a new node and `insertNode` $r\ n\ m$ inserts node m into the data structure as a successor of n. The right hand side shows a possible execution of `compact`. Edges are labeled with their edgesets. The nodes n and m in each iteration are marked in blue. For simplicity, we here assume that the values are identical to their associated timestamps and only show the timestamps.

capacity of nodes grow exponentially with the depth. The right hand side of Figure 9 shows the intermediate states of a potential execution of the `compact` operation.

7.2 High-level proof of `compact`

The verification framework presented in §3 can be easily extended to accommodate any maintenance operation on multicopy structures that does not change the data structure's abstract state. That is, we need to prove that `compact` satisfies the following atomic triple:

$$\langle\, t\, H.\ \mathsf{MCS}(r, t, H)\,\rangle\ \texttt{compact}\ r\ \langle\, \mathsf{MCS}(r, t, H)\,\rangle$$

This specification says that `compact` logically takes effect in a single atomic step, and at this step the abstract state of the data structure does not change. We prove that `compact` satisfies this specification relative to the specifications of the implementation-specific helper functions shown in Figure 10. The postcondition of `mergeContents` is given with respect to an (existentially quantified) set of keys K that are merged from V_n to V_m, resulting in new content sets V_n' and V_m'. The new

$$\{\mathsf{node}(r, n, es_n, V_n)\}\ \ \texttt{atCapacity}\ r\ n\ \ \{b.\, \mathsf{node}(r, n, es_n, V_n)\}$$

$$\{\mathsf{node}(r, n, es_n, V_n)\}$$
$$\texttt{chooseNext}\ r\ n$$
$$\{v.\, \mathsf{node}(r, n, es_n, V_n) * (v = \mathsf{Some}(m) * es_n(m) \neq \emptyset \lor v = \mathsf{None} * \mathsf{needsNewNode}(r, n, es_n, V_n))\}$$

$$\{\mathsf{True}\}\ \ \texttt{allocNode}\ r\ \ \{m.\, \mathsf{node}(r, m, (\lambda n'.\, \emptyset), \emptyset)\}$$

$$\{\mathsf{node}(r, n, es_n, V_n) * \mathsf{needsNewNode}(r, n, es_n, V_n) * \mathsf{node}(r, m, (\lambda n'.\, \emptyset), \emptyset)\}$$
$$\texttt{insertNode}\ r\ n\ m$$
$$\{\mathsf{node}(r, n, es'_n, V_n) * \mathsf{node}(r, m, (\lambda n'.\, \emptyset), \emptyset) * es'_n = es_n[m \rightarrowtail es'_n(m)] * es'_n(m) \neq \emptyset\}$$

$$\{\mathsf{node}(r, n, es_n, V_n) * \mathsf{node}(r, m, es_m, V_m) * es_n(m) \neq \emptyset\}$$
$$\texttt{mergeContents}\ r\ n\ m$$
$$\{\mathsf{node}(r, n, es_n, V'_n) * \mathsf{node}(r, m, es_m, V'_m) * V'_n = \mathit{mergeLeft}(K, V_n, Es, V_m) * V'_m = \mathit{mergeRight}(K, V_n, Es, V_m)\}$$

Fig. 10. Specifications of helper functions used by compact.

contents are determined by the functions *mergeLeft* and *mergeRight* which are defined as follows:

$$\mathit{mergeLeft}(K, V_n, Es, V_m) := \lambda k.\ (k \in K \cap \mathsf{dom}(V_n) \cap Es\ ?\ \bot : V_n(k))$$
$$\mathit{mergeRight}(K, V_n, Es, V_m) := \lambda k.\ (k \in K \cap \mathsf{dom}(V_n) \cap Es\ ?\ V_n(k) : V_m(k))$$

Technically, the linearization point of the operation occurs when all locks are released, just before the function terminates. However, the interesting part of the proof is to show that the changes to the physical contents of nodes n and m performed by each call to mergeContents at line 7 preserve the abstract state of the structure as well as the invariants. In particular, the changes to C_n and C_m also affect the contents-in-reach of m. We need to argue that this is a local effect that does not propagate further in the data structure, as we did in our proof of upsert.

Auxiliary invariants. When proving the correctness of compact, we face two technical challenges. The first challenge arises when establishing that compact changes the contents of the nodes involved in such a way that the high-level invariants are maintained. In particular, we must reestablish Invariant 2, which states that the contents-in-reach of each node can only increase over time. Compaction replaces downstream copies of keys with upstream copies. Thus, in order to maintain Invariant 2, we need the additional auxiliary invariant that the timestamps of keys in the contents of nodes can only decrease as we move away from the root:

Invariant 5 The (timestamp) contents of a node is not smaller than the contents-in-reach of its successor. That is, for all keys k and nodes n and m, if $k \in \mathsf{es}(n, m)$ and $C_n(k) \neq \bot$ then $\mathsf{ts}(C_{ir}(m)(k)) \leqslant \mathsf{ts}(C_n(k))$.

We can capture Invariant 5 in our data structure invariant $\mathsf{MCS}(r, t, H)$ by adding the following predicate as an additional conjunct to the predicate $\mathsf{N}_\mathsf{S}(r, n, C_n, B_n)$:

$$\phi_3(n) := \forall k.\, \mathsf{ts}(Q_n(k)) \leqslant \mathsf{ts}(B_n(k)) \tag{10}$$

The second challenge is that the maintenance template generates only tree-like structures. This implies that at any time there is at most one path from the root to each node in the structure. We will see that this invariant is critical for maintaining Invariant 5. However, the data structure invariant presented thus far allows for arbitrary DAGs.

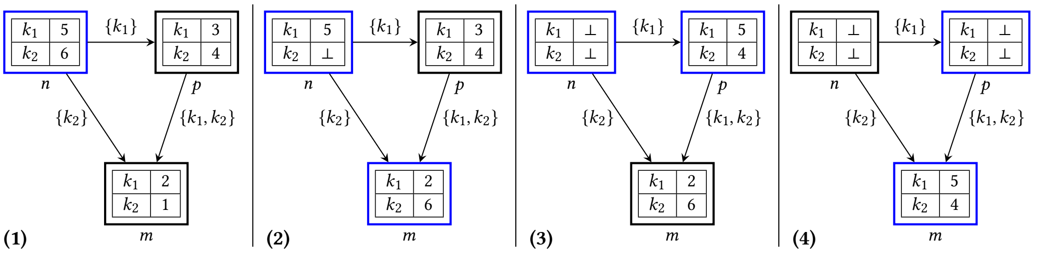

Fig. 11. Possible execution of the compact operation on a DAG. Edges are labeled with their edgesets. The nodes undergoing compaction in each iteration are marked in blue.

To motivate this issue further, consider the multicopy structure in step **(1)** of Figure 11. The logical contents of this structure (i.e. the contents-in-reach of n) is $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (6,6)\}$.

The structure in step **(2)** shows the result obtained after executing compact $r\ n$ to completion where n has been considered to be at capacity and the successor m has been chosen for the merge, resulting in $(k_2, (6,6))$ being moved from n to m. Note that at this point the logical contents of the data structure is still $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (6,6)\}$ as in the original structure. However, the structure now violates Invariant 5 for nodes p and m since $\mathsf{ts}(B_m(k_2)) > \mathsf{ts}(C_p(k_2))$.

Suppose that now a new compaction starts at n that still considers n at capacity and chooses p for the merge. The merge then moves the copy $(k_1, (5,5))$ from n to p. The graph in step **(3)** depicts the resulting structure. The compaction then continues with p, which is also determined to be at capacity. Node m is chosen for the merge, resulting in $(k_1, (5,5))$ and $(k_2, (4,4))$ being moved from p to m. At this point, the second compaction terminates. The final graph in step **(4)** shows the structure obtained at this point. Observe that the logical contents is now $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (4,4)\}$. Thus, this execution violates the specification of compact, which states that the logical contents must be preserved. In fact, a timestamp in the contents-in-reach of n has decreased, which violates Invariant 2.

We observe that although compact will create only tree-like structures, we can prove its correctness using a weaker invariant that does not rule out non-tree DAGs, but instead focuses on how compact interferes with concurrent search operations. This weaker invariant relies on the fact that for every key k in the contents of a node n, there exists a unique search path from the root r to n for k. That is, if we project the graph to only those nodes reachable from the root via edges (n, m) that satisfy $k \in \mathsf{es}(n, m)$, then this projected graph is a list. Using this weaker invariant we can capture implementations based on B-link trees or skip lists which are DAGs but have unique search paths.

To this end, we recall from [Shasha and Goodman 1988] the notion of the *inset* of a node n, $\mathsf{ins}(n)$, which is the set of keys k such that there exists a (possibly empty) path from the root r to n, and k is in the edgeset of all edges along that path. That is, since a search for a key k traverses only those edges (n, m) in the graph that have k in their edgeset, the search traverses (and accesses the contents of) only those nodes n such that $k \in \mathsf{ins}(n)$. Now observe that compact, in turn, moves new copies of a key k downward in the graph only along edges that have k in their edgeset. The following invariant is a consequence of these observations and the definition of contents-in-reach:

Invariant 6 A key is in the contents-in-reach of a node only if it is also in the node's inset. That is, $\mathsf{dom}(C_{ir}(n)) \subseteq \mathsf{ins}(n)$.

This invariant rules out the problematic structure in step **(1)** of Figure 11 because we have $k_2 \in \mathsf{dom}(C_{ir}(p))$ but $k_2 \notin \mathsf{ins}(p) = \{k_1\}$.

Invariant 6 alone is not enough to ensure that Invariant 5 is preserved. For example, consider the structure obtained from **(1)** of Figure 11 by changing the edgeset of the edge (n, p) to $\{k_1, k_2\}$. This modified structure satisfies Invariant 6 but allows the same problematic execution ending in the violation of Invariant 5 that we outlined earlier. However, observe that in the modified structure $k_2 \in \mathsf{es}(n, p) \cap \mathsf{es}(n, m)$, which violates the property that all edgesets leaving a node are disjoint. We have already captured this property in our data structure invariant (as an assumption on the implementation-specific predicate $\mathsf{node}(r, n, es, C_n)$). However, in our formal proof we need to rule out the possibility that a search for k can reach a node m via two *incoming* edgesets $\mathsf{es}(n, m)$ and $\mathsf{es}(p, m)$. Proving that disjoint *outgoing* edgesets imply unique search paths involves global inductive reasoning about the paths in the multicopy structure. To do this using only local reasoning, we will instead rely on an inductive consequence of locally disjoint outgoing edgesets, which we capture explicitly as an additional auxiliary invariant (and which we will enforce using flows):

Invariant 7 The distinct immediate predecessors of any node n have disjoint insets. More precisely, for all distinct nodes n, p, m, and keys k, if $k \in \mathsf{es}(n, m) \cap \mathsf{es}(p, m)$ then $k \notin \mathsf{ins}(n) \cap \mathsf{ins}(p)$.

Note that changing the edgeset of (n, p) in Figure 11 to $\{k_1, k_2\}$ would violate Invariant 7 because the resulting structure would satisfy $k_2 \in \mathsf{es}(n, m) \cap \mathsf{es}(p, m)$ and $k_2 \in \mathsf{ins}(n) \cap \mathsf{ins}(p)$.

In order to capture invariants 6 and 7 in $\mathsf{MCS}(r, t, H)$, we introduce an additional flow that we use to encode the inset of each node. The encoding of insets in terms of a flow follows [Krishna et al. 2020a]. That is, the underlying flow domain is multisets of keys $M = \mathsf{KS} \to \mathbb{N}$ and the actual calculation of the insets is captured by (FlowEqn) if we define:

$$e(n, n') := \lambda m.\, m \cap \mathsf{es}(n, n') \qquad\qquad in(n) := \chi\,(n = r\ ?\ \mathsf{KS} : \emptyset)$$

If $\mathit{fl}_{\mathsf{ins}}$ is a flow that satisfies (FlowEqn) for these definitions of e and in, then for any node n that is reachable from r, $\mathit{fl}_{\mathsf{ins}}(n)(k) > 0$ iff $k \in \mathsf{ins}(n)$. Invariants 6 and 7 are then captured by the following two predicates, which we add to $\mathsf{N_S}$:

$$\phi_4(n) := \forall k.\, k \in \mathsf{dom}(B_n) \Rightarrow \mathit{fl}_{\mathsf{ins}}(n)(k) > 0 \qquad\qquad \phi_5(n) := \forall k.\, \mathit{fl}_{\mathsf{ins}}(n)(k) \leqslant 1$$

Note that ϕ_5 captures Invariant 7 as a property of each individual node n by taking advantage of the fact that the multiset $\mathit{fl}_{\mathsf{ins}}(n)$ explicitly represents all of the contributions made to the inset of n by n's predecessor nodes.

We briefly explain why we can still prove the correctness of `search` and `upsert` with the updated data structure invariant. First note that `search` does not modify the contents, edgesets, or any other ghost resources of any node. So the additional conjuncts in the invariant are trivially maintained.

Now let us consider the operation `upsert` $r\,k\,v$. Since `upsert` does not change the edgesets of any nodes, the resources and constraints related to the inset flow are trivially maintained, with the exception of $\phi_4(r)$: after the `upsert` we have $k \in \mathsf{dom}(B_r)$ which may not have been true before. However, from $in(r)(k) = 1$, the flow equation, and the fact that the flow domain is positive, it follows that we must have $\mathit{fl}_{\mathsf{ins}}(r)(k) > 0$ (i.e., $k \in \mathsf{ins}(r) = \mathsf{KS}$). Hence, $\phi_4(r)$ is preserved as well.

8 PROOF MECHANIZATION

We illustrate the proof methodology presented in this paper by verifying that the multicopy template algorithm (§5, §6, and §7) satisfies search recency. We then instantiate the template to an LSM-like implementation to demonstrate an application of the template. Our proof effort (summarized in Table 1) also contains a mechanically-checked proof that search recency refines the Map ADT specification (§4). We further verify a two-node multicopy structure template that can be instantiated to differential file (DF) structure implementations [Severance and Lohman 1976]. We include this template in our artifact to demonstrate the reuse of the helping proof and because it has a simpler invariant. Though, we provide no implementation for the two-node template. The

Table 1. Summary of templates and instantiations verified in Iris/Coq and GRASShopper. For each algorithm or library, we show the number of lines of code, lines of proof annotation (including specification), total number of lines, and the proof-checking/verification time in seconds.

Templates (Iris/Coq)

Module	Code	Proof	Total	Time
Flow Library	0	3757	3757	41
Lock Implementation	10	352	362	11
Client-level Spec	7	931	938	40
DF Template	19	914	933	90
LSM DAG Template	39	3666	3705	353
Total	**75**	**9620**	**9695**	**535**

Implementations (GRASShopper)

Module	Code	Proof	Total	Time
Array Library	191	440	631	11
LSM Implementation	207	246	453	51
Total	**398**	**686**	**1084**	**62**

artifact is available as a VM image on Zenodo[4] and as source code on GitHub[5]. Verification time was measured on a laptop with an Intel Core i7-8750H CPU and 16GB RAM.

The client-level and template-level proofs were performed in Iris and mechanically verified by the Coq tool, and comprise the left half of Table 1. The flow library formalizes the meta theory of flow interface cameras used in the template proofs. It extends the development of [Krishna et al. 2020a] with a general theory of multiset-based flow domains (about 900 lines).

Our LSM implementation is verified in the SMT-based separation logic tool GRASShopper, and is described in the right half of the table. The implementation uses an unsorted array to store key-timestamp pairs for the (in-memory) root node (with upserts adding to one end of the array), and a read-only sorted array (also known as a sorted string table [Google 2021]) for the other (on-disk) nodes. This array models the contents of a file. The implementation uses a library of utility functions and lemmas for arrays that represent partial maps from keys to values.

We verify both the helper functions for the core search structure operations (Figure 7) as well as those needed by the maintenance template (Figure 9). Each operation demuxes between the code for in-memory and on-disk nodes based on the reference to the operation node. For instance, in the case of `mergeContents` $r\ n\ m$, if $r = n$ then the operation flushes the in-memory node n to the on-disk node m. Otherwise, both n and m must be on-disk nodes, which are then compacted. Alternatively, one could use separate implementations of each helper function for the two types of nodes. The polymorphism could then be resolved statically by unfolding the recursion in the template algorithms once, letting helper function calls in the unfolded iteration go to the in-memory versions and all remaining ones to the on-disk versions.

There are two gaps in the verification that would need to be bridged to obtain a complete end-to-end proof. First, there is currently no way to formally compose the proofs done in Iris/Coq and GRASShopper. However, the two proofs are linked by the node-level specifications of helper functions such as `findNext` at the representation level. As with prior work [Krishna et al. 2020a], we split our verification across two tools in order to take advantage of SMT-based automated techniques for the sequential implementation proofs which are tedious but not technically challenging. Second, GRASShopper does not support reasoning about file access directly. We effectively model each file as a RAM disk whose contents is mapped into memory. This is consistent with the abstract interface for performing file accesses in the LSM tree implementation of LevelDB [Google 2021].

One specific technical challenge that we had to overcome in the Iris formalization is related to the decoupling of the generic client-template proof from the template-implementation proofs. At the linearization point of `upsert`, the proof needs to reestablish the invariant of the helping protocol.

[4] https://zenodo.org/record/5496104
[5] https://github.com/nyu-acsys/template-proofs/tree/multicopy

That is, each template-implementation proof needs to update the relevant ghost resources used for encoding this invariant. We have eliminated the dependency of the template-implementation proof on the concrete representation of the helping protocol invariant by parameterizing this part of the proof over all possible helping protocols that can be maintained by an `upsert`. We discuss this issue in more detail in [Patel et al. 2021, Appendix A.3].

A more desirable solution would be to restrict all reasoning related to the helping protocol to the client-template proof so that the template-implementation proofs do not depend on the helping protocol at all. Essentially, the idea would be to do the relevant ghost state updates in the client-template level proof of `upsert` when the template-level atomic triple of `upsert` is committed. Unfortunately, this idea cannot be realized with Iris' current definition of atomic triples. Proving that the helping protocol invariant is maintained involves the elimination of a so-called *later* modality. That is, one needs to show that a physical computation step is executed at the linearization point (e.g. a memory read or write). However, Iris' atomic triples $\left\langle \vec{x}.\, P \right\rangle\ e\ \left\langle v.\, Q \right\rangle$ are in some sense too abstract, as they do not capture whether c performs a physical computation step. More fine-grained notions of atomic triples are a promising direction for future work.

9 RELATED WORK

Most closely related to our work is the edgeset framework for verifying single-copy structure templates [Krishna et al. 2020a; Shasha and Goodman 1988]. The edgeset framework hinges on the notion of the *keyset* of a node, which is the set of keys that are allowed in the node. That is, a node's contents must be a subset of its keyset. Moreover, the keysets of all nodes must be disjoint. The contribution of Krishna et al. [2020a] is to capture these invariants by a resource algebra in Iris and to show how keysets can be related to the search structure graph using flows to enable local reasoning about template algorithms for single-copy structures. Note that this work [Krishna et al. 2020a; Shasha and Goodman 1988] is limited to single-copy structures since the keyset invariants enforce that every key appears in at most one node. In multicopy structures, the same key may appear in multiple nodes with different associated values.

Relative to [Krishna et al. 2020a; Shasha and Goodman 1988], the main technical novelties are: (i) we identify a node-local quantity (contents-in-reach) for multicopy structures that plays a similar role to the keyset in the single-copy case. Both the invariants that the contents-in-reach must satisfy as well as how the contents-in-reach is encoded using flows is substantially different from the keyset. (ii) We capture the order-preservation aspect of linearizability for multicopy structures in the notion of search recency. (iii) We develop and verify new template algorithms for multicopy structures.

In data structures based on RCU synchronization such as the Citrus tree [Arbel and Attiya 2014], the same key may temporarily appear in multiple nodes. However, such structures are not necessarily multicopy structures. Notably, in a Citrus tree, all copies of a key have the same associated value even in the presence of concurrent updates. Moreover, searches have fixed linearization points. This structure can therefore be handled, in principle, using the single-copy framework of Krishna et al. [2020a] (by building on the formalization of the RCU semantics developed in [Gotsman et al. 2013] and the high-level proof idea for the Citrus tree of Feldman et al. [2020]).

Several other works present generic proof arguments for verifying concurrent traversals of search structures that involve dynamic linearization points [Drachsler-Cohen et al. 2018; Feldman et al. 2018, 2020; O'Hearn et al. 2010]. However, these approaches focus on single-copy structures and rely on global reasoning based on graph reachability.

The idea of tracking auxiliary ghost state about a data structure's history to simplify its linearizability proof has been used in many prior works (e.g. [Bouajjani et al. 2017; Delbianco et al. 2017; Sergey et al. 2015b]). We build on these works and apply this idea to decouple the reasoning about the non-local linearization points of searches from the verification of any specific multicopy structure template.

We have formalized the verification of our template algorithms in Iris [Jung et al. 2018]. Our formalization particularly benefits from Iris's support for user-definable resource algebras, which can capture nontrivial ghost state such as flow interfaces. However, there are a number of other formal proof systems that provide mechanisms for structuring complex linearizability proofs, including other concurrent separation logics [da Rocha Pinto et al. 2014; Dinsdale-Young et al. 2013; Fu et al. 2010; Gardner et al. 2014; Raad et al. 2015; Sergey et al. 2015a] as well as systems based on classical logic [Elmas et al. 2010; Kragl and Qadeer 2018; Kragl et al. 2020]. We also make use of Iris's support for logically atomic triples and prophecy variables to reason modularly about the non-local dynamic linearization points of searches. Specifically, the proof discussed in §4 builds on the prophecy-based Iris proof of the RDCSS data structure from [Jung et al. 2020] and adapts it to a setting where an unbounded number of threads perform "helping". The idea of using prophecy variables to reason about non-fixed linearization points has also been explored in prior work building on logics other than Iris [Sezgin et al. 2010; Vafeiadis 2008; Zhang et al. 2012].

Our proofs rely on both history and prophecy-based reasoning. However, we use the two ideas separately in the two parts of the proof (client-template vs. template-implementation). It does not seem possible to prove the client-template part without using prophecies. The reason is that we use an atomic triple to express the client-level specification. The atomic triple needs to be committed at the actual linearization point. If we were to use only history-based information in the proof, then we would determine at the point when $(k, (v, t'))$ is found that the linearization point already happened in the past. However, at that point, it is already too late to commit the atomic triple.

A proof that uses only history-based verification and does not rely on atomic triples is likely possible. For instance, one alternative approaches to using atomic triples is to prove that the template-level atomic specification contextually refines the client-level atomic specification of multicopy structures using a relational program logic. A number of prior works have developed such refinement-based approaches [Banerjee et al. 2016; Frumin et al. 2018, 2020], including for settings that involve unbounded helping [Liang and Feng 2013; Turon et al. 2013]. An alternative approach to using prophecy variables for reasoning about non-fixed linearization points is to explicitly construct a partial order of events as the program executes, effectively representing all the possible linearizations that are consistent with the observations made so far [Khyzha et al. 2017].

There has also been much work on obtaining fully automated proofs of linearizability by static analysis and model checking [Abdulla et al. 2013, 2018; Amit et al. 2007; Bouajjani et al. 2013, 2015, 2017; Cerný et al. 2010; Dragoi et al. 2013; Henzinger et al. 2013; Lesani et al. 2014; Vafeiadis 2009; Zhu et al. 2015]. The proof framework presented in this paper is capable of reasoning about implementations that are beyond the reach of current automatic techniques, via interactive (though still machine-checked) template proofs. We hope that this framework will help to inform the design of future automated static analyzers for concurrent programs.

Multicopy structures such as the LSM tree are often used in file and database systems to organize data that spans multiple storage media, e.g., RAM and hard disks. Several prior projects have considered the formal verification of file systems. SibyllFS [Ridge et al. 2015] provides formal specifications for POSIX-based file system implementations to enable systematic testing of existing implementations. FSCQ [Chen et al. 2015], Yggdrasil [Bornholt et al. 2016; Sigurbjarnarson et al. 2016], and DFSCQ [Chen et al. 2017] provide formally verified file system implementations that also guarantee crash consistency. However, these implementations do not support concurrent execution of file system operations. Our work provides a framework for reasoning about the in-memory concurrency aspects of multicopy structures. However, we mostly abstract from issues related to the interaction with the different storage media. Notably, in our verified LSM tree implementation, we do not model disk failure and hence do not address crash consistency.

Distributed key/value stores have to contend with copies of keys being present in multiple nodes at a time. Several works verify consistency of operations performed on such data structures [Chordia et al. 2013; Kaki et al. 2018; Xiong et al. 2020], including linearizability [Wang et al. 2019]. In the distributed context, the main technical challenge arises from data replication and the ensuing weakly consistent semantics of concurrent operations. As we consider lock-based templates, we can assume a sequentially consistent memory model for our verification. For lock-free multicopy structures such as the Bw-tree [Levandoski et al. 2013], weak memory consistency may be a concern. Lock-free multicopy structures also require the development of new template algorithms, which then need to be shown linearizable with respect to the template-level specification. However, once this is established, linearizability with respect to the client-level specification is obtained for free. We also believe that the high-level invariants from §6 are applicable towards proving the template-level specification. For instance, each lock-free node-local list of the Bw-tree behaves like a multicopy structure and satisfies the identified invariants.

10 CONCLUSION

This paper and the accompanying verification effort have made the following contributions: We presented a general framework for verifying concurrent multicopy structures. The framework introduces an intermediate abstraction layer that enables reasoning about concurrent multicopy structures in terms of template algorithms that abstract from the data structure representation. We constructed such a template algorithm that generalizes the log-structured merge tree to DAGs and proved its correctness. The proof is decomposed into two parts to maximize proof reuse: (1) a general reduction of linearizability of multicopy structures that eliminates the need to reason about non-local linearization points; and (2) a generic proof of the template algorithm that abstracts from the data structure's memory representation in concrete implementations. The full proof is formalized in the concurrent separation logic Iris and mechanized in Coq. We have also verified an instantiation of the template algorithm to LSM trees, resulting in the first formally-verified concurrent multicopy search structure.

ACKNOWLEDGMENTS

This work is funded in parts by the National Science Foundation under grants 1925605, 1815633, 1934388, 1840761, and 1339362. Further funding came from NYU WIRELESS and from the New York University Abu Dhabi Center for Interacting Urban Networks (CITIES). We thank Elizabeth Dietrich and Raphael Sofaer for their help on mechanizing the proofs of the differential file template. We also extend our gratitude to the anonymous reviewers of OOPSLA'21 whose questions helped us clarify the presentation. We would also like to thank the reviewers of the book [Krishna et al. 2021], specifically Maurice Herlihy, Eddie Kohler, Robbert Krebbers, K. Rustan M. Leino, and Peter Müller, for their suggestions to improve the presentation.

REFERENCES

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.* IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezine. 2013. An Integrated Specification and Verification Technique for Highly Concurrent Data Structures. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 324–338. https://doi.org/10.1007/978-3-642-36742-7_23

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. 2018. Fragment Abstraction for Concurrent Shape Analysis. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 442–471. https://doi.org/10.1007/978-3-319-89884-1_16

Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

Apache Software Foundation. 2021. Apache Cassandra. https://cassandra.apache.org/. Last accessed on August 12, 2021.

Maya Arbel and Hagit Attiya. 2014. Concurrent updates with RCU: search tree as an example. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, Magnús M. Halldórsson and Shlomi Dolev (Eds.). ACM, 196–205. https://doi.org/10.1145/2611462.2611471

Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. 2016. Relational Logic with Framing and Hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India (LIPIcs, Vol. 65)*, Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:16. https://doi.org/10.4230/LIPIcs.FSTTCS.2016.11

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 83–98. https://doi.org/10.1145/2872362.2872406

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. On Reducing Linearizability to State Reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9135)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. 2017. Proving Linearizability Using Forward Simulations. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 542–563. https://doi.org/10.1007/978-3-319-63390-9_28

Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. 2010. Model Checking of Linearizability of Concurrent List Implementations. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 465–479. https://doi.org/10.1007/978-3-642-14295-6_41

Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Mert Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* ACM, 270–286. https://doi.org/10.1145/3132747.3132776

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 18–37. https://doi.org/10.1145/2815400.2815402

Sagar Chordia, Sriram K. Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. 2013. Asynchronous Resilient Linearizability. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8205)*, Yehuda Afek (Ed.). Springer, 164–178. https://doi.org/10.1007/978-3-642-41527-2_12

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A.

Bernstein (Eds.). ACM, 505–520. https://doi.org/10.1145/3183713.3196927

Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2017. Concurrent Data Structures Linked in Time. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:30. https://doi.org/10.4230/LIPIcs.ECOOP.2017.8

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. 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, 287–300. https://doi.org/10.1145/2429069.2429104

Dana Drachsler-Cohen, Martin T. Vechev, and Eran Yahav. 2018. Practical concurrent traversals in search trees. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 207–218. https://doi.org/10.1145/3178487.3178503

Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. 2013. Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 174–190. https://doi.org/10.1007/978-3-642-39799-8_11

Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. 2010. Simplifying Linearizability Proofs with Reduction and Abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6015)*, Javier Esparza and Rupak Majumdar (Eds.). Springer, 296–311. https://doi.org/10.1007/978-3-642-12002-2_25

Manuel Fahndrich and Rustan Leino. 2003. Heap Monotonic Typestate. In *Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO)* (proceedings of the first international workshop on alias confinement and ownership (iwaco) ed.). https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/

Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. 2018. Order out of Chaos: Proving Linearizability Using Local Views. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018 (LIPIcs, Vol. 121)*, Ulrich Schmid and Josef Widder (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:21. https://doi.org/10.4230/LIPIcs.DISC.2018.23

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 128:1–128:29. https://doi.org/10.1145/3428196

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *CoRR* abs/2006.13635 (2020). arXiv:2006.13635 https://arxiv.org/abs/2006.13635

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6269)*, Paul Gastin and François Laroussinie (Eds.). Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Philippa Gardner, Azalea Raad, Mark J. Wheelhouse, and Adam Wright. 2014. Abstract Local Reasoning for Concurrent Libraries: Mind the Gap. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 308)*, Bart Jacobs, Alexandra Silva, and Sam Staton (Eds.). Elsevier, 147–166. https://doi.org/10.1016/j.entcs.2014.10.009

Google. 2021. LevelDB. https://github.com/google/leveldb. Last accessed on August 12, 2021.

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 249–269. https://doi.org/10.1007/978-3-642-37036-6_15

Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8052)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.). Springer, 242–256. https://doi.org/10.1007/978-3-642-40184-8_18

Maurice Herlihy and J. D. Tygar. 1987. How to Make Replicated Data Secure. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 293)*, Carl Pomerance (Ed.). Springer, 379–391. https:

//doi.org/10.1007/3-540-48184-2_33

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

Bart Jacobs and Frank Piessens. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 271–282. https://doi.org/10.1145/1926385.1926417

Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 377–396. https://doi.org/10.1007/978-3-642-28869-2_19

Jonathan Ellis. 2011. Leveled Compaction in Apache Cassandra. https://www.datastax.com/blog/2011/10/leveled-compaction-apache-cassandra. Last accessed on August 12, 2021.

Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 321–332.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2018. Alone together: compositional reasoning and inference for weak isolation. *Proc. ACM Program. Lang.* 2, POPL (2018), 27:1–27:34. https://doi.org/10.1145/3158115

Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10201)*, Hongseok Yang (Ed.). Springer, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 79–102. https://doi.org/10.1007/978-3-319-96145-3_5

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 275–298. https://doi.org/10.1007/978-3-030-53288-8_14

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. 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, 205–217. https://doi.org/10.1145/3009837.3009855

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2021. *Automated Verification of Concurrent Search Structures.* Morgan & Claypool Publishers. https://doi.org/10.2200/S01089ED1V01Y202104CSL013

Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. 2020a. Verifying concurrent search structure templates. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 181–196. https://doi.org/10.1145/3385412.3386029

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020b. Local Reasoning for Global Graph Properties. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture*

Notes in Computer Science, Vol. 12075), Peter Müller (Ed.). Springer, 308–335. https://doi.org/10.1007/978-3-030-44914-8_12

Mohsen Lesani, Todd D. Millstein, and Jens Palsberg. 2014. Automatic Atomicity Verification for Clients of Concurrent Data Structures. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 550–567. https://doi.org/10.1007/978-3-319-08867-9_37

Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2491956.2462189

Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *VLDB J.* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, Andréa W. Richa and Rachid Guerraoui (Eds.). ACM, 85–94. https://doi.org/10.1145/1835698.1835722

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying Concurrent Multicopy Search Structures. *CoRR* abs/2109.05631 (2021). arXiv:2109.05631 http://arxiv.org/abs/2109.05631

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 124–139. https://doi.org/10.1007/978-3-642-54862-8_9

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 710–735. https://doi.org/10.1007/978-3-662-46669-8_29

Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 497–514. https://doi.org/10.1145/3132747.3132765

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 38–53. https://doi.org/10.1145/2815400.2815411

Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 217–228. https://doi.org/10.1145/2213836.2213862

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015a. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 77–87. https://doi.org/10.1145/2737924.2737964

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015b. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 333–358. https://doi.org/10.1007/978-3-662-46669-8_14

Dennis G. Severance and Guy M. Lohman. 1976. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.* 1, 3 (1976), 256–267. https://doi.org/10.1145/320473.320484

Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. 2010. Tressa: Claiming the Future. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 25–39. https://doi.org/10.1007/978-3-642-15057-9_2

Dennis E. Shasha and Nathan Goodman. 1988. Concurrent Search Structure Algorithms. *ACM Trans. Database Syst.* 13, 1 (1988), 53–90. https://doi.org/10.1145/42201.42204

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

Risi Thonangi and Jun Yang. 2017. On Log-Structured Merge for Solid-State Drives. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 683–694. https://doi.org/10.1109/ICDE.2017.121

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained 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, 343–356. https://doi.org/10.1145/2429069.2429111

Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

Viktor Vafeiadis. 2009. Shape-Value Abstraction for Verifying Linearizability. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5403)*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer, 335–348. https://doi.org/10.1007/978-3-540-93900-9_27

Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 980–993. https://doi.org/10.1145/3314221.3314617

Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, Shan Lu and Erik Riedel (Eds.). USENIX Association, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

Shale Xiong, Andrea Cerone, Azalea Raad, and Philippa Gardner. 2020. Data Consistency in Transactional Storage Systems: A Centralised Semantics. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:31. https://doi.org/10.4230/LIPIcs.ECOOP.2020.21

Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A Structural Approach to Prophecy Variables. In *Theory and Applications of Models of Computation - 9th Annual Conference, TAMC 2012, Beijing, China, May 16-21, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7287)*, Manindra Agrawal, S. Barry Cooper, and Angsheng Li (Eds.). Springer, 61–71. https://doi.org/10.1007/978-3-642-29952-0_12

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2015. Poling: SMT Aided Linearizability Proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 3–19. https://doi.org/10.1007/978-3-319-21668-3_1