Fragment Abstraction for Concurrent Shape Analysis

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh

Uppsala University

Abstract. A major challenge in automated verification is to develop techniques that are able to reason about fine-grained concurrent algorithms that consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded dynamically allocated memory. Existing automated techniques consider the case where shared data is organized into singly-linked lists. We present a novel shape analysis for automated verification of fine-grained concurrent algorithms that can handle heap structures which are more complex than just singly-linked lists, in particular skip lists and arrays of singly linked lists, while at the same time handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap. Our technique is based on a novel shape abstraction, which represents a set of heaps by a set of *fragments*. A fragment is an abstraction of a pair of heap cells that are connected by a pointer field. We have implemented our approach and applied it to automatically verify correctness, in the sense of linearizability, of most linearizable concurrent implementations of sets, stacks, and queues, which employ singly-linked lists, skip lists, or arrays of singly-linked lists with timestamps, which are known to us in the literature.

1 Introduction

Concurrent algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state are of central importance in a large number of software systems. They provide efficient concurrent realizations of common interface abstractions, and are widely used in libraries, such as the Intel Threading Building Blocks or the `java.util.concurrent` package. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking when possible. A number of bugs in published algorithms have been reported [13, 30]. Consequently, significant research efforts have been directed towards developing techniques to verify correctness of such algorithms. One widely-used correctness criterion is that of *linearizability*, meaning that each method invocation can be considered to occur atomically at some point between its call and return. Many of the developed verification techniques require significant *manual* effort for constructing correctness proofs (e.g., [25, 41]), in some cases with the support of an interactive theorem prover (e.g., [40, 11, 35]). Development of automated verification techniques remains a difficult challenge.

A major challenge for the development of automated verification techniques is that such techniques must be able to reason about fine-grained concurrent algorithms that are infinite-state in many dimensions: they consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded

dynamically allocated memory. Perhaps the hardest of these challenges is that of handling dynamically allocated memory. Consequently, existing techniques that can automatically prove correctness of such fine-grained concurrent algorithms restrict attention to the case where heap structures represent shared data by singly-linked lists [1, 18, 3, 36, 42]. Furthermore, many of these techniques impose additional restrictions on the considered verification problem, such as bounding the number of accessing threads [4, 45, 43]. However, in many concurrent data structure implementations the heap represents more sophisticated structures, such as skiplists [16, 22, 38] and arrays of of singly-linked lists [12]. There are no techniques that have been applied to automatically verify concurrent algorithms that operate on such data structures.

Contributions In this paper, we present a technique for automatic verification of concurrent data structure implementations that operate on dynamically allocated heap structures which are more complex than just singly-linked lists. Our framework is the first that can automatically verify concurrent data structure implementations that employ singly linked lists, skiplists [16, 22, 38], as well as arrays of singly linked lists [12], at the same time as handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap.

Our technique is based on a novel shape abstraction, called *fragment abstraction*, which in a simple and uniform way is able to represent several different classes of unbounded heap structures. Its main idea is to represent a set of heap states by a set of *fragments*. A fragment represents two heap cells that are connected by a pointer field. For each of its cells, the fragment represents the contents of its non-pointer fields, together with information about how the cell can be reached from the program's global pointer variables. The latter information consists of both: (i) *local* information, saying which pointer variables point directly to them, and (ii) *global* information, saying how the cell can reach to and be reached from (by following chains of pointers) heap cells that are globally significant, typically since some global variable points to them. A set of fragments represents the set of heap states in which any two pointer-connected nodes is represented by some fragment in the set. Thus, a set of fragments describes the set of heaps that can be formed by "piecing together" fragments in the set. The combination of local and global information in fragments supports reasoning about the sequence of cells that can be accessed by threads that traverse the heap by following pointer fields in cells and pointer variables: the local information captures properties of the cell fields that can be accessed as a thread dereferences a pointer variable or a pointer field; the global information also captures whether certain significant accesses will at all be possible by following a sequence of pointer fields. This support for reasoning about patterns of cell accesses enables automated verification of reachability and other functional properties.

Fragment abstraction can (and should) be combined, in a natural way, with data abstractions for handling unbounded data domains and with thread abstractions for handling an unbounded number of threads. For the latter we adapt the successful thread-modular approach [5], which represents the local state of a single, but arbitrary thread, together with the part of the global state and heap that is accessible to that thread. Our combination of fragment abstraction, thread abstraction, and data abstraction results in a finite abstract domain, thereby guaranteeing termination of our analysis.

We have implemented our approach and applied it to automatically verify correctness, in the sense of linearizability, of a large number of concurrent data structure algorithms, described in a C-like language. More specifically, we have automatically verified linearizability of most linearizable concurrent implementations of sets, stacks, and queues, and priority queues, which employ singly-linked lists, skiplists, or arrays of timestamped singly-linked lists, which are known to us in the literature on concurrent data structures. For this verification, we specify linearizability using the simple and powerful technique of *observers* [1, 7, 9], which reduces the criterion of linearizability to a simple reachability property. To verify implementations of stacks and queues, the application of observers can be done completely automatically without any manual steps, whereas for implementations of sets, the verification relies on light-weight user annotation of how linearization points are placed in each method [3],.

The fact that our fragment abstraction has been able to automatically verify all supplied concurrent algorithms, also those that employ skiplists or arrays of SLLs, indicates that the fragment abstraction is a simple mechanism for capturing both the local and global information about heap cells that is necessary for verifying correctness, in particular for concurrent algorithms where an unbounded number of threads interact via a shared heap.

Outline In the next section, we illustrate our fragment abstraction on the verification of a skiplist-based concurrent set implementation. In Section 3 we introduce our model for programs, and of observers for specifying linearizability. In Section 4 we describe in more detail our fragment abstraction for skiplists; note that singly-linked lists can be handled as a simple special case of skiplists. In Section 5 we describe how fragment abstraction applies to arrays of singly-linked lists with timestamp fields. Our implementation and experiments are reported in Section 6, followed by conclusions in Section 7.

Related Work A large number of techniques have been developed for representing heap structures in automated analysis, including, e.g., separation logic and various related graph formalisms [47, 10, 15], other logics [33], automata [23], or graph grammars [19]. Most works apply these to sequential programs.

Approaches for automated verification of concurrent algorithms are limited to the case of singly-linked lists [1, 18, 3, 36, 42]. Furthermore, many of these techniques impose additional restrictions on the considered verification problem, such as bounding the number of accessing threads [4, 45, 43].

In [1], concurrent programs operating on SLLs are analyzed using an adaptation of a transitive closure logic [6], combined with tracking of simple sortedness properties between data elements; the approach does not allow to represent patterns observed by threads when following sequences of pointers inside the heap, and so has not been applied to concurrent set implementations. In our recent work [3], we extended this approach to handle SLL implementations of concurrent sets by adapting a well-known abstraction of singly-linked lists [28] for concurrent programs. The resulting technique is specifically tailored for singly-links. Our fragment abstraction is significantly simpler conceptually, and can therefore be adapted also for other classes of heap structures. The approach of [3] is the only one with a shape representation strong enough to verify concurrent set implementations based on sorted and non-sorted singly-linked lists having

non-optimistic contains (or lookup) operations we consider, such as the lock-free sets of *HM* [22], *Harris* [17], or *Michael* [29], or unordered set of [48]. As shown in Section 6, our fragment abstraction can handle them as well as also algorithms employing skiplists and arrays of singly-linked lists.

There is no previous work on automated verification of skiplist-based concurrent algorithms. Verification of *sequential* algorithms have been addressed under restrictions, such as limiting the number of levels to two or three [23, 2]. The work [34] generates verification conditions for statements in sequential skiplist implementations. All these works assume that skiplists have the well-formedness property that any higher-level lists is a sublist of any lower-level list, which is true for sequential skiplist algorithms, but false for several concurrent ones, such as [22, 26].

Concurrent algorithms based on arrays of SLLs, and including timestamps, e.g., for verifying the algorithms in [12] have shown to be rather challenging. Only recently has the TS stack been verified by non-automated techniques [8] using a non-trivial extension of forward simulation, and the TS queue been verified manually by a new technique based on partial orders [24, 37].. We have verified both these algorithms automatically using fragment abstraction,

Our fragment abstraction is related in spirit to other formalisms that abstract dynamic graph structures by defining some form of equivalence on its nodes (e.g., [46, 33, 23]). These have been applied to verify functional correctness fine-grained concurrent algorithms for a limited number of SLL-based algorithms. Fragment abstraction's representation of both local and global information allows to extend the applicability of this class of techniques.

2 Overview

In this section, we illustrate our technique on the verification of correctness, in the sense of linearizability, of a concurrent set data structure based on skiplists, namely the Lock-Free Concurrent Skiplist from [22, Section 14.4]. Skiplists provide expected logarithmic time search while avoiding some of the complications of tree structures. Informally, a skiplist consists of a collection of sorted linked lists, each of which is located at a *level*, ranging from 1 up to a maximum value. Each skiplist node has a key value and participates in the lists at levels 1 up to its *height*. The skiplist has sentinel head and tail nodes with maximum heights and key values $-\infty$ and $+\infty$, respectively. The lowest-level list (at level 1) constitutes an ordered list of all nodes in the skiplist. Higher-level lists are increasingly sparse sublists of the lowest-level list, and serve as shortcuts into lower-level lists. Figure 1 shows an example of a skiplist of height 3. It has head and tail nodes of height 3, two nodes of height 2, and one node of height 1.

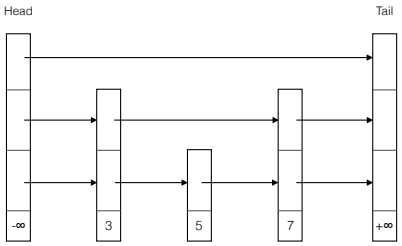


Fig. 1: An example of skiplist

The algorithm has three main methods, namely `add`, `contains` and `remove`. The method `add(x)` adds `x` to the set and returns true iff `x` was not already in the set; `remove(x)` removes `x` from the set and returns true iff `x` was in the set; and `contains(x)` returns true iff `x` is in the set. All methods rely on a method `find` to search for a given key. In this section, we shortly describe the `find` and `add` methods. Figure 2 shows code for these two methods.

```
struct Node { int key; int height; Node next[]; boolean marked[];}
```

```
boolean find(int x,Node preds[],Node succs[])
 1  boolean marked = false;
 2  boolean s;
 3  retry:
 4  while (true)
 5    pred = H;
 6    for (int k = MAXLEVEL; k >= 1; k--)
 7      curr = pred.next[k];
 8      while (true)
 9        <succ, marked> =
                 <curr.next[k], curr.marked[k]>;
10        while (marked)
11          s=CAS(<pred.next[k],pred.marked[k]>
                    ,<curr,false>,<succ,false>);
12          if (!s) goto retry;
13          curr = pred.next[k]; •
14          <succ, marked =
              <curr.next[k], curr.marked[k]>;
15        if (curr.key < x)
16          pred = curr;
17          curr = succ;
18        else break;
19      preds[k] = pred;
20      succs[k] = curr;
21    return (curr.key == x);
```

```
boolean add (int x):
 1  int h = randomLevel;
 2  Node* preds[1..h]; succs[1..h]
 3  while (true);
 4    if find(x,preds,succs)
 5      return false;
 6    else
 7      Node* n = new Node(x, h);
 8      for (int k = 1;k <= h; k++)
 9        <n.next[k],n.marked[k]> =
                          <succ[k],false>;
10      Node* pred = preds[1];
11      Node* succ = succs[1];
12      <n.next[1],n.marked[1]>=<succ,false>
13      if !CAS(<pred.next[1],pred.marked[1]>
                 ,<succ,false>,<n,false>);
14        goto 3;
15      else •
16        for (int k = 2; k <= h; k++)
17          while (true);
18          pred = preds[k];
19          succ = succs[k];
20          if CAS(<pred.next[k],pred.marked
                   [k]>,<succ,false>,<n,false>)
21            break;
22          find(x,preds,succs);
23  return true;
```

Fig. 2: Code for the `find` and `add` methods of the skiplist algorithm.

In the algorithm, each heap node has a `key` field, a `height`, an array of `next` pointers indexed from 1 up to its `height`, and an array of `marked` fields which are true if the node has been logically removed at the corresponding level. Removal of a node (at a certain level `k`) occurs in two steps: first the node is logically removed by setting its `marked` flag at level `k` to `true`, thereafter the node is physically removed by unlinking it from the level-`k` list. The algorithm must be able to update the `next`[`k`] pointer and `marked`[`k`] field together as one atomic operation; this is standardly implemented by encoding them in a single word. The head and tail nodes of the skiplist are pointed to by global pointer variables `H` and `T`, respectively. The `find` method traverses the list at decreasing levels using two local variables `pred` and `curr`, starting at the head and at the maximum level (lines 5-6). At each level `k` it sets `curr` to `pred.next`[`k`] (line 7). During the traversal, the pointer variable `succ` and boolean variable `marked` are atom-

ically assigned the values of curr.next[k] and curr.marked[k], respectively (line 9, 14). After that, the method repeatedly removes marked nodes at the current level (lines 10 to 14). This is done by using a CompareAndSwap (CAS) command (line 11), which tests whether pred.next[k] and pred.marked[k] are equal to curr and false respectively. If this test succeeds, it replaces them with succ and false and returns true; otherwise, the CAS returns false. During the traversal at level k, pred and curr are advanced until pred points to a node with the largest key at level k which is smaller than x (lines 15-18). Thereafter, the resulting values of pred and curr are recorded into preds[k] and succs[k] (lines 19, 20), whereafter traversal continues one level below until it reaches the bottom level. Finally, the method returns true if the key value of curr is equal to x; otherwise, it returns false meaning that a node with key x is not found.

The add method uses find to check whether a node with key x is already in the list. If so it returns false; otherwise, a new node is created with randomly chosen height h (line 7), and with next pointers at levels from 1 to h initialised to corresponding elements of succ (line 8 to 9). Thereafter, the new node is added into the list by linking it into the bottom-level list between the preds[1] and succs[1] pointers returned by find. This is achieved by using a CAS to make preds[1].next[1] point to the new node (line 13). If the CAS fails, the add method will restart from the beginning (line 3) by calling find again, etc. Otherwise, add proceeds with linking the new node into the list at increasingly higher levels (lines 16 to 22). For each higher level k, it makes preds[k].next[k] point to the new node if it is still valid (line 20); otherwise find is called again to recompute preds[k] and succs[k] on the remaining unlinked levels (line 22). Once all levels are linked, the method returns true.

To prepare for verification, we add a specification which expresses that the skiplist algorithm of Figure 2 is a linearizable implementation of a set data structure, using the technique of *observers* [1, 7, 9, 3]. For our skiplist algorithm, the user first instruments statements in each method that correspond to linearization points (LPs), so that their execution announces the corresponding atomic set operation. In Figure 2, the LP of a successful add operation is at line 15 of the add method (denoted by a blue dot) when the CAS succeeds, whereas the LP of an unsuccessful add operation is at line 13 of the find method (denoted by a red dot). We must now verify that in any concurrent execution of a collection of method calls, the sequence of announced operations satisfies the semantics of the set data structure. This check is performed by an *observer*, which monitors the sequence of announced operations. The observer for the set data structure utilizes a register, which is initialized with a single, arbitrary key value. It checks that operations on this particular value follow set semantics, i.e., that successful add and remove operations on an element alternate and that contains are consistent with them. We form the cross-product of the program and the observer, synchronizing on operation announcements. This reduces the problem of checking linearizability to the problem of checking that in this cross-product, regardless of the initial observer register value, the observer cannot reach a state where the semantics of the set data structure has been violated.

To verify that that the observer cannot reach a state where a violation is reported, we compute a symbolic representation of an invariant that is satisfied by all reachable con-

figurations of the cross-product of a program and an observer. This symbolic representation combines thread abstraction, data abstraction and our novel *fragment abstraction* to represent the heap state. Our *thread abstraction* adapts the thread-modular approach by representing only the view of single, but arbitrary, thread `th`. Such a view consists of the local state of thread `th`, including the value of the program counter, the state of the observer, and the part of the heap that is accessible to thread `th` via pointer variables (local to `th` or global). Our *data abstraction* represents variables and cell fields that range over small finite domains by their concrete values, whereas variables and fields that range over the same domain as `key` fields are abstracted to constraints over their relative ordering (wrp. to $<$).

In our *fragment abstraction*, we represent the part of the heap that is accessible to thread `th` by a set of *fragments*. A fragment represents a pair of heap cells (accessible to `th`) that are connected by a pointer field, under the applied data abstraction. A fragment is a triple of form $\langle \mathtt{i}, \mathtt{o}, \phi \rangle$, where `i` and `o` are *tags* that represent the two cells, and ϕ is a subset of $\{<, =, >\}$ which constrains the order between the `key` fields of the cells. Each tag is a tuple $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, where

- `dabs` represents the non-pointer fields of the cell under the applied data abstraction,
- `pvars` is the set of (local to `th` or global) pointer variables that point to the cell,
- `reachfrom` is the set of (i) global pointer variables from which the cell represented by the tag is reachable via a (possibly empty) sequence of `next`$[1]$ pointers, and (ii) observer registers $\mathtt{x}_i$ such that the cell is reachable from some cell whose data value equals that of $\mathtt{x}_i$,
- `reachto` is the corresponding information, but now considering cells that are reachable from the cell represented by the tag.
- `private` is `true` only if $\mathbb{c}$ is private to `th`.

Thus, the fragment contains both (i) *local* information about the cell's fields and variables that point to it, as well as (ii) *global* information, representing how each cell in the pair can reach to and be reached from (by following a chain of pointers) a small set of globally significant heap cells.

A set of fragments represents the set of heap structures in which each pair of pointer-connected nodes is represented by some fragment in the set. Put differently, a set of fragments describes the set of heaps that can be formed by "piecing together" pairs of pointer-connected nodes that are represented by some fragment in the set. This "piecing together" must be both locally consistent (appending only fragments that agree on their common node), and globally consistent (respecting the global reachability information).

When applying fragment abstraction to skiplists, we use two types of fragments: *level 1-fragments* for nodes connected by a `next`$[1]$-pointer, and *higher level-fragments* for nodes connected by a higher level pointer. In other words, we abstract all levels higher than 2 by the abstract element `higher`. Thus, a pointer or non-pointer variable of form `v`$[\mathtt{k}]$, indexed by a level $\mathtt{k} \geq 2$, is abstracted to `v`$[\mathtt{higher}]$.

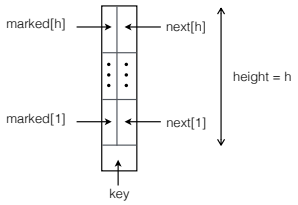


Fig. 3: A structure of a cell

Let us illustrate how fragment abstraction applies to the skiplist algorithm. Figure 4

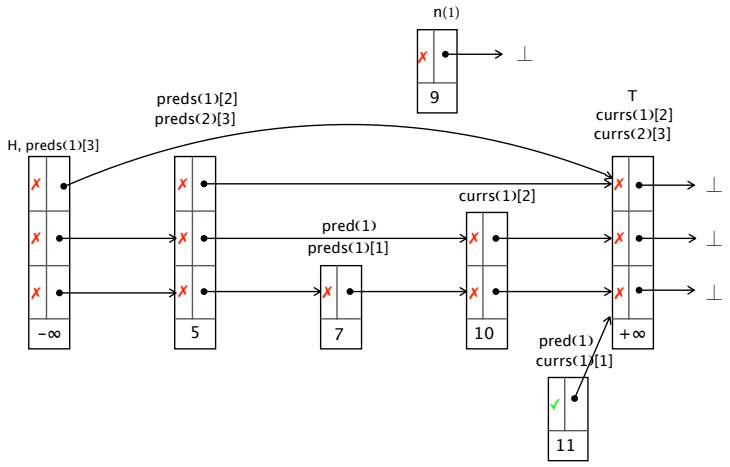


Fig. 4: A heap shape of a 3-level skiplist with two threads active

shows an example heap state of the skiplist algorithm with three levels. Each heap cell is shown with the values of its fields as described in Figure 3. In addition, each cell is labeled by the pointer variables that point to it; we use `preds(i)[k]` to denote the local variable `preds[k]` of thread $\texttt{th}_i$, and the same for other local variables. In the heap state of Figure 4, thread $\texttt{th}_1$ is trying to add a new node of height 1 with key 9, and has reached line 8 of the `add` method. Thread $\texttt{th}_2$ is trying to add a new node with key 20 and it has done its first iteration of the `for` loop in the `find` method. The variables `preds(2)[3]` and `currs(2)[3]` have been assigned so that the new node (which has not yet been created) will be inserted between node 5 and the tail node. The observer is not shown, but the value of the observer register is 9; thus it currently tracks the `add` operation of $\texttt{th}_1$.

Figure 5 illustrates how pairs of heap nodes can be represented by fragments. As a first example, in the view of thread $\texttt{th}_1$, the two left-most cells in Figure 4 are represented by the level 1-fragment $\texttt{v}_1$ in Figure 5. Here, the variable `preds(1)[3]` is represented by `preds[higher]`. The mapping π_1 represents the data abstraction of the `key` field, here saying that it is smaller than the value 9 of the observer register. The two left-most cells are also represented by a higher-level fragment, viz. v_8. The pair consisting of the two sentinel cells (with keys $-\infty$ and $+\infty$) is represented by the higher-level fragment v_9. In each fragment, the abstraction `dabs` of non-pointer fields are shown represented inside each tag of the fragment. The ϕ is shown as a label on the arrow between two tags. Above each tag is `pvars`. The first row under each tag is `reachfrom`, whereas the second row is `reachto`.

Figure 5 shows a set of fragments that is sufficient to represent the part of the heap that is accessible to $\texttt{th}_1$ in the configuration in Figure 4. There are 11 fragments, named $\texttt{v}_1, \ldots, \texttt{v}_{11}$. Two of these ($\texttt{v}_6$, $\texttt{v}_7$ and $\texttt{v}_{11}$) consist of a tag that points to $\perp$. All other fragments consist of a pair of pointer-connected tags. The fragments $\texttt{v}_1, \ldots, \texttt{v}_6$ are level-1-fragments, whereas $\texttt{v}_7, \ldots, \texttt{v}_{11}$ are higher level-fragments. The `private` field

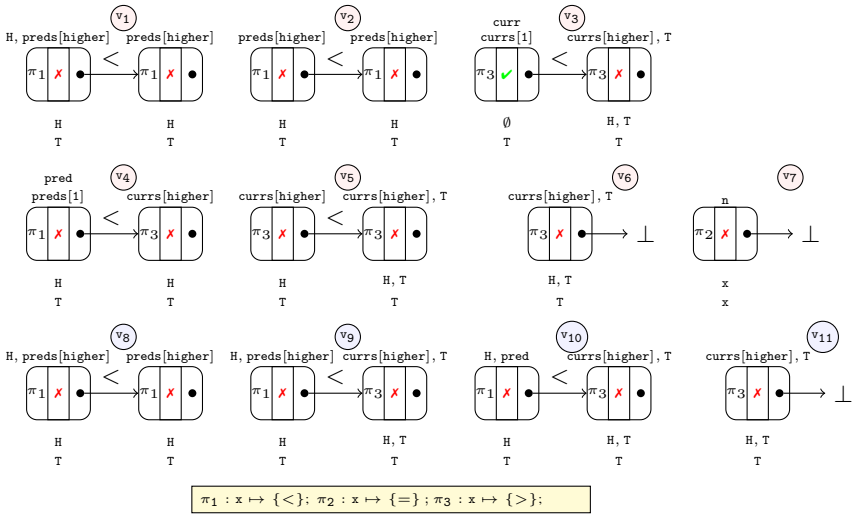


Fig. 5: Fragment abstraction of skiplist algorithm

of the input tag of v_7 is `true`, whereas the `private` field of tags of other fragments are `false`.

To verify linearizability of the algorithm in Figure 2, we must represent several key invariants of the heap. These include (among others):

1. the bottom-level list is strictly sorted in `key` order,
2. a higher-level pointer from a globally reachable node is a shortcut into the level-1 list, i.e., it points to a node that is reachable by a sequence of `next`[1] pointers,
3. all nodes which are unreachable from the head of the list are marked, and
4. the variable `pred` points to a cell whose `key` field is never larger than the input parameter of its `add` method.

Let us illustrate how such invariants are captured by our fragment abstraction. 1) All level-1 fragments are strictly sorted, implying that the bottom-level list is strictly sorted. 2) For each higher-level fragment `v`, if $\texttt{H} \in \texttt{v.i.reachfrom}$ then also $\texttt{H} \in \texttt{v.o.reachfrom}$, implying (together with $\texttt{v}.\phi = \{<\}$) that the cell represented by `v.o` it is reachable from that represented by `v.i` by a sequence of `next`[1]-pointers. 3) This is verified by inspecting each tag: v_3 contains the only unreachable tag, and it is also marked. 4) The fragments express this property in the case where the value of `key` is the same as the value of the observer register `x`. Since the invariant holds for any value of `x`, this property is sufficiently represented for purposes of verification.

3 Concurrent Data Structure Implementations

In this section, we introduce our representation of concurrent data structure implementations, we define the correctness criterion of linearizability, we introduce observers and how to use them for specifying linearizability.

3.1 Concurrent Data Structure Implementations

We first introduce (sequential) data structures. A *data structure* DS is a pair $\langle \mathbb{D}, \mathbb{M} \rangle$, where $\mathbb{D}$ is a (possibly infinite) *data domain* and $\mathbb{M}$ is an alphabet of *method names*. An *operation op* is of the form $\mathtt{m}(d^{in}, d^{out})$, where $\mathtt{m} \in \mathbb{M}$ is a method name, and d^{in}, d^{out} are the *input* resp. *output* values, each of which is either in $\mathbb{D}$ or in some small finite domain $\mathbb{F}$, which includes the booleans. For some method names, the input or output value is absent from the operation. A *trace* of DS is a sequence of operations. The (sequential) semantics of a data structure DS is given by a set $[\![\mathtt{DS}]\!]$ of allowed traces. For example, a Set data structure has method names add, remove, and contains. An example of an allowed trace is $\mathtt{add}(3, \mathtt{true})\ \mathtt{contains}(4, \mathtt{false})\ \mathtt{contains}(3, \mathtt{true})\ \mathtt{remove}(3, \mathtt{true})$.

A *concurrent data structure implementation* operates on a shared state consisting of shared global variables and a shared heap. It assigns, to each method name, a method which performs operations on the shared state. It also comes with a method named init, which initializes its shared state.

A *heap (state)* $\mathcal{H}$ consists of a finite set $\mathbb{C}$ of cells, including the two special cells null and $\perp$ (dangling). Heap cells have a fixed set $\mathcal{F}$ of fields, namely non-pointer fields that assume values in $\mathbb{D}$ or $\mathbb{F}$, and possibly lock fields. We use the term $\mathbb{D}$-*field* for a non-pointer field that assumes values in $\mathbb{D}$, and the terms $\mathbb{F}$-*field* and *lock field* with analogous meaning. Furthermore, each cell has one or several named pointer fields. For instance, in data structure implementations based on singly-linked lists, each heap cell has a pointer field named next; in implementations based on skiplists there is an array of pointer fields named next[k] where k ranges from 1 to a maximum level.

Each method declares local variables and a method body. The set of local variables includes the input parameter of the method and the program counter pc. A *local state* loc of a thread th defines the values of its local variables. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. Variables are either pointer variables (to heap cells), locks, or data variables assuming values in $\mathbb{D}$ or $\mathbb{F}$. We assume that all global variables are pointer variables. The body is built in the standard way from atomic commands, using standard control flow constructs (sequential composition, selection, and loop constructs). Atomic commands include assignments between variables, or fields of cells pointed to by a pointer variable. Method execution is terminated by executing a return command, which may return a value. The command new Node() allocates a new structure of type Node on the heap, and returns a reference to it. The compare-and-swap command $\mathtt{CAS}(\mathtt{a}, \mathtt{b}, \mathtt{c})$ atomically compares the values of a and b. If equal, it assigns the value of c to a and returns true, otherwise, it leaves a unchanged and returns false. We assume a memory management mechanism, which automatically collects garbage, and ensures that a new cell is fresh, i.e., has not been used before; this avoids the so-called ABA problem (e.g., [31]).

We define a *program* $\mathcal{P}$ (over a concurrent data structure) to consist of an arbitrary number of concurrently executing threads, each of which executes a method that performs an operation on the data structure. The shared state is initialized by the init method prior to the start of program execution. A *configuration* of a program $\mathcal{P}$ is a tuple $c_{\mathcal{P}} = \langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle$ where T is a set of threads, $\mathcal{H}$ is a heap, and LOC maps each thread

$\mathtt{th} \in \mathtt{T}$ to its local state $\mathtt{LOC}(\mathtt{th})$. We assume concurrent execution according to sequentially consistent memory model. The behavior of a thread $\mathtt{th}$ executing a method can be formalized as a transition relation $\rightarrow_{\mathtt{th}}$ on pairs $\langle \mathtt{loc}, \mathcal{H} \rangle$ consisting of a local state $\mathtt{loc}$ and a heap state $\mathcal{H}$. The behavior of a program $\mathcal{P}$ can be formalized by a transition relation $\rightarrow_{\mathcal{P}}$ on program configurations; each step corresponds to a move of a single thread. I.e., there is a transition of form $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle \rightarrow_{\mathcal{P}} \langle \mathtt{T}, \mathtt{LOC}[\mathtt{th} \leftarrow \mathtt{loc}'], \mathcal{H}' \rangle$ whenever some thread $\mathtt{th} \in \mathtt{T}$ has a transition $\langle \mathtt{loc}, \mathcal{H} \rangle \rightarrow_{\mathtt{th}} \langle \mathtt{loc}', \mathcal{H}' \rangle$ with $\mathtt{LOC}(\mathtt{th}) = \mathtt{loc}$.

3.2 Linearizability

In a concurrent data structure implementation, we represent the calling of a method by a *call action* $\mathtt{call}_{\mathtt{o}}\ \mathtt{m}\left(d^{in}\right)$, and the return of a method by a *return action* $\mathtt{ret}_{\mathtt{o}}\ \mathtt{m}\left(d^{out}\right)$, where $\mathtt{o} \in \mathbb{N}$ is an *action identifier*, which links the call and return of each method invocation. A *history* h is a sequence of actions such that (i) different occurrences of return actions have different action identifiers, and (ii) for each return action a_2 in h there is a unique *matching* call action a_1 with the same action identifier and method name, which occurs before a_2 in h. A call action which does not match any return action in h is said to be *pending*. A history without pending call actions is said to be *complete*. A *completed extension* of h is a complete history h' obtained from h by appending (at the end) zero or more return actions that are matched by pending call actions in h, and thereafter removing the call actions that are still pending. For action identifiers $\mathtt{o}_1, \mathtt{o}_2$, we write $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ to denote that the return action with identifier $\mathtt{o}_1$ occurs before the call action with identifier $\mathtt{o}_2$ in h. A complete history is *sequential* if it is of the form $a_1 a_1' a_2 a_2' \cdots a_n a_n'$ where a_i' is the matching action of a_i for all $i : 1 \leq i \leq n$, i.e., each call action is immediately followed by its matching return action. We identify a sequential history of the above form with the corresponding trace $op_1\, op_2 \cdots op_n$ where $op_i = \mathtt{m}(d_i^{in}, d_i^{out})$, $a_i = \mathtt{call}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{in}\right)$, and $a_i = \mathtt{ret}_{\mathtt{o}_i}\ \mathtt{m}\left(d_i^{out}\right)$, i.e., we merge each call action together with the matching return action into one operation. A complete history h' is a *linearization* of h if (i) h' is a permutation of h, (ii) h' is sequential, and (iii) $\mathtt{o}_1 \preceq_{\mathtt{h}'} \mathtt{o}_2$ if $\mathtt{o}_1 \preceq_{\mathtt{h}} \mathtt{o}_2$ for each pair of action identifiers $\mathtt{o}_1$ and $\mathtt{o}_2$. A sequential history h' is *valid* wrt. $\mathtt{DS}$ if the corresponding trace is in $[\![\mathtt{DS}]\!]$. We say that h is *linearizable* wrt. $\mathtt{DS}$ if there is a completed extension of h, which has a linearization that is valid wrt. $\mathtt{DS}$. We say that a program $\mathcal{P}$ is linearizable wrt. $\mathtt{DS}$ if, in each possible execution, the sequence of call and return actions is *linearizable* wrt. $\mathtt{DS}$.

We specify linearizability using the technique of *observers* [1, 7, 9, 3]. Depending on the data structure, we apply it in two different ways.

- For implementations of sets and priority queues, the user instruments each method so that it announces a corresponding operation precisely when the method executes its LP, either directly or with lightweight instrumentation using the technique of linearization policies [3]. We represent such announcements by labels on the program transition relation $\rightarrow_{\mathcal{P}}$, resulting in transitions of form $c_{\mathcal{P}} \xrightarrow{\mathtt{m}(d^{in}, d^{out})}_{\mathcal{P}} c'_{\mathcal{P}}$. Thereafter, an *observer* is constructed, which monitors the sequence of operations that is announced by the instrumentation; it reports (by moving to an accepting error location) whenever this sequence violates the (sequential) semantics of the data structure.

– For stacks and queues, we use a recent result [7, 9] that the set of linearizable histories, i.e., sequences of call and return actions, can be exactly specified by an observer. Thus, linearizability can be specified without any user-supplied instrumentation, by using an observer which monitors the the sequences of call and return actions and reports violations of linearizability.

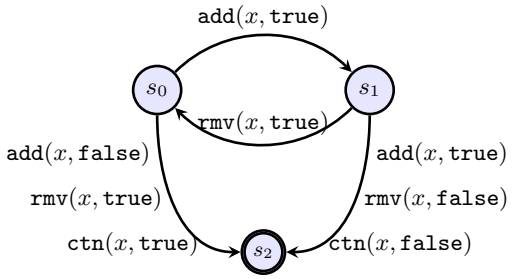


Fig. 6: Set observer.

Formally, an observer $\mathcal{O}$ is a tuple $\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\texttt{init}}, \texttt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\texttt{acc}} \rangle$ where $S^{\mathcal{O}}$ is a finite set of *observer locations* including the *initial location* $s^{\mathcal{O}}_{\texttt{init}}$ and the *accepting location* $s^{\mathcal{O}}_{\texttt{acc}}$, a finite set $\texttt{X}^{\mathcal{O}}$ of *registers*, and $\Delta^{\mathcal{O}}$ is a finite set of *transitions*. For observers that monitor sequences of operations, transitions are of the form $\langle s_1, \texttt{m}(x^{in}, x^{out}), s_2 \rangle$, where $\texttt{m} \in \mathbb{M}$ is a method name and x^{in} and x^{out} are either registers or constants, i.e., transitions are labeled by operations whose input or output data may be parameterized on registers. The observer processes a sequence of operations one operation at a time. If there is a transition, whose label (after replacing registers by their values) matches the operation, such a transition is performed. If there is no such transition, the observer remains in its current location. The observer accepts a sequence if it can be processed in such a way that an accepting location is reached. The observer is defined in such a way that it accepts precisely those sequences that are *not* in $[\![\texttt{DS}]\!]$. Fig. 6 depicts an observer for the set data structure.

To check that no execution of the program announces a sequence of labels that can drive the observer to an accepting location, we form the cross-product $\mathcal{S} = \mathcal{P} \otimes \mathcal{O}$ of the program $\mathcal{P}$ and the observer $\mathcal{O}$, synchronizing on common transition labels. Thus, configurations of $\mathcal{S}$ are of the form $\langle c_{\mathcal{P}}, \langle s, \rho \rangle \rangle$, consisting of a program configuration $c_{\mathcal{P}}$, an observer location s, and an assignment ρ of values in $\mathbb{D}$ to the observer registers. Transitions of $\mathcal{S}$ are of the form $\langle c_{\mathcal{P}}, \langle s, \rho \rangle \rangle \rightarrow_{\mathcal{S}} \langle {c_{\mathcal{P}}}', \langle s', \rho \rangle \rangle$, obtained from a transition $c_{\mathcal{P}} \xrightarrow{\lambda}_{\mathcal{P}} {c_{\mathcal{P}}}'$ of the program with some (possibly empty) label λ, where the observer makes a transition $s \xrightarrow{\lambda} s'$ if it can perform such a matching transition, otherwise $s' = s$. Note that the observer registers are not changed. We also add straightforward instrumentation to check that each method invocation announces exactly one operation, whose input and output values agree with the method's parameters and return value. This reduces the problem of checking linearizability to the problem of checking that in this cross-product, the observer cannot reach an accepting error location.

4 Verification using Fragment Abstraction for Skiplists

In the previous section, we reduced the problem of verifying linearizability to the problem of verifying that, in any execution of the cross-product of a program and an observer, the observer cannot reach an accepting location. We perform this verification by computing a symbolic representation of an invariant that is satisfied by all reachable configurations of the cross-product, using an abstract interpretation-based fixpoint procedure, starting from a symbolic representation of the set of initial configurations, thereafter repeatedly performing symbolic postcondition computations that extend the symbolic representation by the effect of any execution step of the program, until convergence.

In Section 4.1, we define in more detail our symbolic representation for skiplists, focusing in particular on the use of fragment abstraction, and thereafter (in Section 4.2) describe the symbolic postcondition computation. Since singly-linked lists is a trivial special case of skiplists, we can use the relevant part of this technique also for programs based on singly-linked lists.

4.1 Symbolic Representation

This subsection contains a more detailed description of our symbolic representation for programs that operate on skiplists, which was introduced in Section 2. We first describe the data abstraction, thereafter the fragment abstraction, and finally their combination into a symbolic representation.

Data Abstraction Our data abstraction is defined by assigning a abstract domain to each concrete domain of data values, as follows.

- For small concrete domains (including that of the program counter, and of the observer location), the abstract domain is the same as the concrete one.
- For locks, the abstract domain is $\{me, other, free\}$, meaning that the lock is held by the concerned thread, held by some other thread, or is free, respectively.
- For the concrete domain $\mathbb{D}$ of data values, the abstract domain is the set of mappings from observer registers and local variables ranging over $\mathbb{D}$ to subsets of $\{<, =, >\}$. An mapping in this abstract domain represents the set of data values $\mathtt{d}$ such that it maps each local variable and observer register with a value $\mathtt{d}' \in \mathbb{D}$ to a set which includes a relation $\sim$ such that $\mathtt{d} \sim \mathtt{d}'$.

Fragment Abstraction Let us now define our fragment abstraction for skiplists. For presentation purposes, we assume that each heap cell has at most one $\mathbb{D}$-field, named `data`. For an observer register $\mathtt{x}_i$, let a $\mathtt{x}_i$*-cell* be a heap cell whose `data` field has the same value as $\mathtt{x}_i$.

Since the number of levels is unbounded, we define an abstraction for levels. Let $\mathtt{k}$ be a level. Define the abstraction of a pointer variable of form $\mathtt{p}[\mathtt{k}]$, denoted $\widehat{\mathtt{p}[\mathtt{k}]}$, to be $\mathtt{p}[1]$ if $\mathtt{k} = 1$, and to be $\mathtt{p}[\mathtt{higher}]$ if $\mathtt{k} \geq 2$. That is, this abstraction does not distinguish different higher levels.

A *tag* is a tuple $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, where (i) `dabs` is a mapping from non-pointer fields to their corresponding abstract domains;

if a non-pointer field is an array indexed by levels, then the abstract domain is that for single elements: e.g., the abstract domain for the array `marked` in Figure 2 is simply the set of booleans, (ii) `pvars` is a set of abstracted pointer variables, (iii) `reachfrom` and `reachto` are sets of global pointer variables and observer registers, and (iv) `private` is a boolean value.

For a heap cell $\mathbb{c}$ that is accessible to thread `th` in a configuration $c_{\mathcal{S}}$, and a tag $\mathtt{tag} = \langle \mathtt{dabs}, \mathtt{pvars}, \mathtt{reachfrom}, \mathtt{reachto}, \mathtt{private} \rangle$, we let $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{tag}$ denote that $\mathbb{c}$ satisfies the tag `tag` "at level `k`". More precisely, this means that

- `dabs` is an abstraction of the concrete values of the non-pointer fields of $\mathbb{c}$; for array fields `f` we use the concrete value $\mathtt{f}[\mathtt{k}]$,
- `pvars` is the set of abstractions of pointer variables (global or local to `th`) that point to $\mathbb{c}$,
- `reachfrom` is the set of (i) abstractions of global pointer variables from which $\mathbb{c}$ is reachable via a (possibly empty) sequence of $\mathtt{next}[1]$ pointers, and (ii) observer registers $\mathtt{x}_i$ such that $\mathbb{c}$ is reachable from some $\mathtt{x}_i$-cell (via a sequence of $\mathtt{next}[1]$ pointers),
- `reachto` is the set of (i) abstractions of global pointer variables pointing to a cell that is reachable (via a sequence of $\mathtt{next}[1]$ pointers) from $\mathbb{c}$, and (ii) observer registers $\mathtt{x}_i$ such that some $\mathtt{x}_i$-cell is reachable from $\mathbb{c}$.
- `private` is `true` only if $\mathbb{c}$ is not accessible to any other thread than `th`.

Note that the global information represented by the fields `reachfrom` and `reachto` concerns *only* reachability via level-1 pointers.

A *skiplist fragment* `v` (or just fragment) is a triple of form $\langle \mathtt{i}, \mathtt{o}, \phi \rangle$, of form $\langle \mathtt{i}, \mathtt{null} \rangle$, or of form $\langle \mathtt{i}, \bot \rangle$, where `i` and `o` are tags and ϕ is a subset of $\{<, =, >\}$. Each skiplist fragment additionally has a *type*, which is either *level-1* or *higher-level* (note that a level-1 fragment can otherwise be identical to a higher-level fragment). For a cell $\mathbb{c}$ which is accessible to thread `th`, and a fragment `v` of form $\langle \mathtt{i}, \mathtt{o}, \phi \rangle$, let $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{v}$ denote that the $\mathtt{next}[\mathtt{k}]$ field of $\mathbb{c}$ points to a cell $\mathbb{c}'$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{i}$, and $\mathbb{c}' \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{o}$, and $\mathbb{c}.\mathtt{data} \sim \mathbb{c}'.\mathtt{data}$ for some $\sim \in \phi$. The definition of $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{v}$ is adapted to fragments of form $\langle \mathtt{i}, \mathtt{null} \rangle$ and $\langle \mathtt{i}, \bot \rangle$ in the obvious way. For a fragment $\mathtt{v} = \langle \mathtt{i}, \mathtt{o}, \phi \rangle$, we often use $\mathtt{v}.\mathtt{i}$ for `i` and $\mathtt{v}.\mathtt{o}$ for `o`, etc.

Let V be a set of fragments. A global configuration $c_{\mathcal{S}}$ satisfies V wrp. to `th`, denoted $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}} V$, if

- for any cell $\mathbb{c}$ that is accessible to `th` (different from `null` and $\bot$), there is a level-1 fragment $\mathtt{v} \in V$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},1} \mathtt{v}$, and
- for all levels `k` from 2 up to the height of $\mathbb{c}$, there is a higher-level fragment $\mathtt{v} \in V$ such that $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}} \mathtt{v}$.

Intuitively, a set of fragment represents the set of heap states, in which each pair of cells connected by a $\mathtt{next}[1]$ pointer is represented by a level-1 fragment, and each pair of cells connected by a $\mathtt{next}[\mathtt{k}]$ pointer for $\mathtt{k} \geq 2$ is represented by a higher-level fragment which represents array fields of cells at index `k`.

Symbolic Representation We can now define our abstract symbolic representation.

Define a *local symbolic configuration* σ to be a mapping from local non-pointer variables (including the program counter) to their corresponding abstract domains. We let $c_{\mathcal{S}} \models^{loc}_{\mathtt{th}} \sigma$ denote that in the global configuration $c_{\mathcal{S}}$, the local configuration of thread th satisfies the local symbolic configuration σ, defined in the natural way. For a local symbolic configuration σ, an observer location s, a pair V of fragments and a thread th, we write $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, V \rangle$ to denote that (i) $c_{\mathcal{S}} \models^{loc}_{\mathtt{th}} \sigma$, (ii) the observer is in location s, and (iii) $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}} V$.

Definition 1. *A* symbolic representation Ψ *is a partial mapping from pairs of local symbolic configurations and observer locations to sets of fragments. A system configuration* $c_{\mathcal{S}}$ *satisfies a symbolic representation* Ψ*, denoted* $c_{\mathcal{S}}$ *sat* Ψ*, if for each thread* th*, the domain of* Ψ *contains a pair* $\langle \sigma, s \rangle$ *such that* $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, \Psi(\langle \sigma, s \rangle) \rangle$.

4.2 Symbolic Postcondition Computation

The symbolic postcondition computation must ensure that the symbolic representation of the reachable configurations of a program is closed under execution of a statement by some thread. That is, given a symbolic representation Ψ, the symbolic postcondition operation must produce an extension Ψ' of Ψ, such that whenever $c_{\mathcal{S}}$ sat Ψ and $c_{\mathcal{S}} \rightarrow_{\mathcal{S}} c'_{\mathcal{S}}$ then ${c_{\mathcal{S}}}'$ sat Ψ'. Let th be an arbitrary thread. Then $c_{\mathcal{S}}$ sat Ψ means that $Dom(\Psi)$ contains some pair $\langle \sigma, s \rangle$ with $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, \Psi(\langle \sigma, s \rangle) \rangle$. The symbolic postcondition computation must ensure that $Dom(\Psi')$ contains a pair $\langle \sigma', s' \rangle$ such that $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', \Psi'(\langle \sigma', s' \rangle) \rangle$. In the thread-modular approach, there are two cases to consider, depending on which thread causes the step from $c_{\mathcal{S}}$ to ${c_{\mathcal{S}}}'$.

- *Local Steps:* The step is caused by th itself executing a statement which may change its local state, the location of the observer, and the state of the heap. In this case, we first compute a local symbolic configuration σ', an observer location s', and a set V' of fragments such that $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', V' \rangle$, and then (if necessary) extend Ψ so that $\langle \sigma', s' \rangle \in Dom(\Psi)$ and $V' \subseteq \Psi(\langle \sigma', s' \rangle)$.
- *Interference Steps:* The step is caused by another thread $\mathtt{th}_2$, executing a statement which may change the location of the observer (to s') and the heap. By $c_{\mathcal{S}}$ sat Ψ there is a local symbolic configuration σ_2 with $\langle \sigma_2, s \rangle \in Dom(\Psi)$ such that $c_{\mathcal{S}} \models_{\mathtt{th}_2} \langle \sigma_2, s, \Psi(\langle \sigma_2, s \rangle) \rangle$. For any such σ_2 and statement of $\mathtt{th}_2$, we must compute a set V' of fragments such that the resulting configuration ${c_{\mathcal{S}}}'$ satisfies $c'_{\mathcal{S}} \models^{heap}_{\mathtt{th}} V'$ and ensure that $\langle \sigma, s' \rangle \in Dom(\Psi)$ and $V' \subseteq \Psi(\langle \sigma, s' \rangle)$. To do this, we first combine the local symbolic configurations σ and σ_2 and the sets of fragments $\Psi(\langle \sigma, s \rangle)$ and $\Psi(\langle \sigma_2, s \rangle)$, using an operation called *intersection*, into a joint local symbolic configuration of th and $\mathtt{th}_2$ and a set $V_{1,2}$ of fragments that represents the cells accessible to either th or $\mathtt{th}_2$. We thereafter symbolically compute the postcondition of the statement executed by $\mathtt{th}_2$, in the same was as for local steps, and finally project the set of resulting fragments back onto th to obtain V'.

In the following, we first describe the symbolic postcondition computation for local steps, and thereafter the intersection operation.

Symbolic Postcondition Computation for Local Steps Let th be an arbitrary thread, assume that $\langle \sigma, s \rangle \in Dom(\Psi)$, and let $V = \Psi(\langle \sigma, s \rangle)$ For each statement that th can execute in a configuration $c_{\mathcal{S}}$ with $c_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma, s, V \rangle$, we must compute a local symbolic configuration σ', a new observer location s' and a set V' of fragments such that the resulting configuration ${c_{\mathcal{S}}}'$ satisfies $c'_{\mathcal{S}} \models_{\mathtt{th}} \langle \sigma', s', V' \rangle$. This computation is done differently for each statement. For statements that do not affect the heap or pointer variables, this computation is standard, and affects only the local symbolic configuration, the observer location, and the dabs component of tags. We therefore here describe how to compute the effect of statements that update pointer variables or pointer fields of heap cells, since these are the most interesting cases. In this computation, the set V' is constructed in two steps: (1) First, the level-1 fragments of V' are computed, based on the level-1 fragments in V. (2) Thereafter, the higher-level fragments of V' are computed, based on the higher-level fragments in V and how fragments in V are transformed when entered in to V'. We first describe the construction of level-1 fragments, and thereafter the construction of higher-level fragments.

Construction of level-1 fragments Let us first intuitively introduce techniques used for constructing the level-1 fragments of V'. Consider a statement of form g := p, which assigns the value of a local pointer variable p to a global pointer variable g. The set V' of fragments is obtained by modifying fragments in V to reflect the effect of the assignment. For any tag in a fragment, the dabs field is not affected. The pvars field is updated to contain the variable g if and only if it contained the variable p before the statement. The difficulty is to update the reachability information represented by the fields reachfrom and reachto, and in particular to determine whether g should be in such a set after the statement (note that if p were a global variable, then the corresponding reachability information for p would be in the fields reachfrom and reachto, and the update would be simple, reflecting that g and p become aliases). In order to construct V' with sufficient precision, we therefore investigate whether the set of fragments V allows to form a heap in which a p-cell can reach or be reached from (by a sequence of next[1] pointers) a particular tag of a fragment. We also investigate whether a heap can be formed in which a p-cell can *not* reach or be reached from a particular tag. For each such successful investigation, the set V' will contain a level-1 fragment with corresponding contents of its reachto and reachfrom fields.

The postcondition computation performs this investigation by computing a set of transitive closure-like relations between level-1 fragments, which represent reachability via sequences of next[1] pointers (since only these are relevant for the reachfrom and reachto fields). First, say that two tags tag and tag′ are *consistent* (wrp. to a set of fragments V) if the concretizations of their dabs-fields overlap, and if the other fields pvars, reachfrom, reachto, and private) agree. Thus, tag and tag′ are consistent if there can exist a cell $\mathbb{c}$ accessible to th in some heap, with $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th}} \mathtt{tag}$ and $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\mathtt{th}} \mathtt{tag}'$. Next, for two level-1 fragments $\mathtt{v}_1$ and $\mathtt{v}_2$ in a set V of fragments,

- let $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o}$ and $\mathtt{v}_2.\mathtt{i}$ are consistent, and
- let $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denote that $\mathtt{v}_1.\mathtt{o} = \mathtt{v}_2.\mathtt{o}$ are consistent, and that either $\mathtt{v}_1.\mathtt{i}.\mathtt{pvars} \cap \mathtt{v}_2.\mathtt{i}.\mathtt{pvars} = \emptyset$ or the global variables in $\mathtt{v}_1.\mathtt{i}.\mathtt{reachfrom}$ are disjoint from those in $\mathtt{v}_2.\mathtt{i}.\mathtt{reachfrom}$.

Intuitively, $\mathtt{v}_1 \hookrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathbb{c}_1.\mathtt{next}[1] = \mathbb{c}_2$ for some cells with $\mathbb{c}_1 \lhd^{\mathcal{CS}}_{\mathtt{th},1} \mathtt{v}_1$ and $\mathbb{c}_2 \lhd^{\mathcal{CS}}_{\mathtt{th},1} \mathtt{v}_2$. Intuitively, $\mathtt{v}_1 \leftrightarrow_V \mathtt{v}_2$ denotes that it is possible that $\mathbb{c}_1.\mathtt{next}[1] = \mathbb{c}_2.\mathtt{next}[1]$ for different cells $\mathbb{c}_1$ and $\mathbb{c}_2$ with $\mathbb{c}_1 \lhd^{\mathcal{CS}}_{\mathtt{th},1} \mathtt{v}_1$ and $\mathbb{c}_2 \lhd^{\mathcal{CS}}_{\mathtt{th},1} \mathtt{v}_2$ (Note that these definitions also work for fragments containing `null` or $\bot$). We use these relations to define the following derived relations on level-1 fragments:

- $\overset{+}{\hookrightarrow}_V$ denotes the transitive closure, and $\overset{*}{\hookrightarrow}_V$ the reflexive transitive closure, of $\hookrightarrow_V$,
- $\mathtt{v}_1 \overset{**}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{*}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{*\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{*}{\hookrightarrow}_V \mathtt{v}'_1$,
- $\mathtt{v}_1 \overset{++}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1, \mathtt{v}'_2 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}'_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$ and $\mathtt{v}_2 \overset{+}{\hookrightarrow}_V \mathtt{v}'_2$,
- $\mathtt{v}_1 \overset{+\circ}{\leftrightarrow}_V \mathtt{v}_2$ denotes that $\exists \mathtt{v}'_1 \in V$ with $\mathtt{v}'_1 \leftrightarrow_V \mathtt{v}_2$ where $\mathtt{v}_1 \overset{+}{\hookrightarrow}_V \mathtt{v}'_1$.

We sometimes use, e.g., $\mathtt{v}_2 \overset{+*}{\leftrightarrow}_V \mathtt{v}_1$ for $\mathtt{v}_1 \overset{*+}{\leftrightarrow}_V \mathtt{v}_2$. We say that $\mathtt{v}_1$ and $\mathtt{v}_2$ are *compatible* if $\mathtt{v}_x \overset{*}{\hookrightarrow} \mathtt{v}_y$, or $\mathtt{v}_y \overset{*}{\hookrightarrow} \mathtt{v}_x$, or $\mathtt{v}_x \overset{**}{\leftrightarrow} \mathtt{v}_y$. Intuitively, if $\mathtt{v}_1$ and $\mathtt{v}_2$ are satisfied by two cells in the same heap state, then they must be compatible.

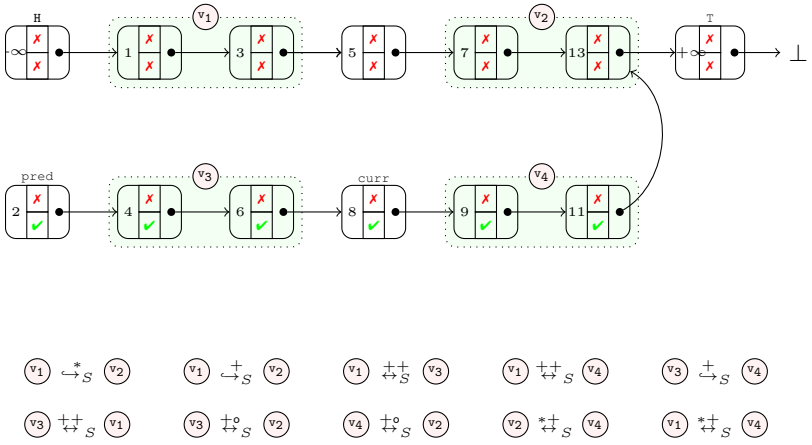


Fig. 7: Illustration of some transitive closure-like relations between fragments

Figure 7 illustrates the above relations for a heap state with 13 heap cells. The figure depicts, in green, four pairs of heap cells connected by a $\mathtt{next}[1]$ pointer, which satisfy the four fragments $\mathtt{v}_1$, $\mathtt{v}_2$, $\mathtt{v}_3$, and $\mathtt{v}_4$, respectively. At the bottom are depicted the transitive-closure like relations that hold between these fragments.

We can now describe the symbolic postcondition computation for statements that affect pointer variables or fields. This is a case analysis, and for space reasons we only include some representative cases.

First, consider a statement of form $\mathtt{x} := \mathtt{y}$, where $\mathtt{x}$ and $\mathtt{y}$ are local (to thread $\mathtt{th}$) or global pointer variables. We must compute a set V' of fragments which are satisfied

by the configuration after the statement. We first compute the level-1-fragments in V' as follows (higher-level fragments will be computed later). We observe that for any cell $\mathbb{c}$ which is accessible to $\mathtt{th}$ after the statement, there must be some level-1 fragment $\mathtt{v}'$ in V' with $\mathbb{c} \lhd^{c_S}_{\mathtt{th},1} \mathtt{v}'$. By assumption, $\mathbb{c}$ satisfies some fragment $\mathtt{v}$ in V before the statement, and is in the same heap state as the cell pointed to by $\mathtt{y}$. This implies that $\mathtt{v}$ must be compatible with some fragment $\mathtt{v}_y \in V$ such that $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i}.\mathtt{pvars}$ (recall that $\widehat{\mathtt{y}}$ is the abstraction of $\mathtt{y}$, which in the case that $\mathtt{y}$ is an array element maps higher level indices to that abstract index $\mathtt{higher}$). This means that we can make a case analysis on the possible relationships between $\mathtt{v}$ and any such $\mathtt{v}_y$. Thus, for each fragment $\mathtt{v}_y \in V$ such that $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i}.\mathtt{pvars}$ we let V' contain the fragments obtained by any of the following transformations on any fragment in V.

1. First, for the fragment $\mathtt{v}_y$ itself, we let V' contain $\mathtt{v}'_y$, which is the same as $\mathtt{v}_y$, except that
   - $\mathtt{v}'_y.\mathtt{i}.\mathtt{pvars} = \mathtt{v}_y.\mathtt{i}.\mathtt{pvars} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{o}.\mathtt{pvars} = \mathtt{v}.\mathtt{o}.\mathtt{pvars} \setminus \{\widehat{\mathtt{x}}\}$

   and furthermore, if $\mathtt{x}$ is a global variable, then
   - $\mathtt{v}'_y.\mathtt{i}.\mathtt{reachto} = \mathtt{v}_y.\mathtt{i}.\mathtt{reachto} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}_y.\mathtt{i}.\mathtt{reachfrom} \cup \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'_y.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}_y.\mathtt{o}.\mathtt{reachfrom} \cup \{\widehat{\mathtt{x}}\}$ and $\mathtt{v}'_y.\mathtt{o}.\mathtt{reachto} = \mathtt{v}_y.\mathtt{o}.\mathtt{reachto} \setminus \{\widehat{\mathtt{x}}\}$.
2. for each $\mathtt{v}$ with $\mathtt{v} \hookrightarrow_V \mathtt{v}_y$, let V' contain $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that
   - $\mathtt{v}'.\mathtt{i}.\mathtt{pvars} = \mathtt{v}.\mathtt{i}.\mathtt{pvars} \setminus \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{pvars} = \mathtt{v}.\mathtt{o}.\mathtt{pvars} \cup \{\widehat{\mathtt{x}}\}$,
   - $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{i}.\mathtt{reachfrom} \setminus \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{i}.\mathtt{reachto} = \mathtt{v}.\mathtt{i}.\mathtt{reachto} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}.\mathtt{o}.\mathtt{reachfrom} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
   - $\mathtt{v}'.\mathtt{o}.\mathtt{reachto} = \mathtt{v}.\mathtt{o}.\mathtt{reachto} \cup \{\widehat{\mathtt{x}}\}$ if $\mathtt{x}$ is a global variable,
3. We perform analogous inclusions for fragments $\mathtt{v}$ with $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$, $\mathtt{v}_y \overset{*}{\hookrightarrow}_V \mathtt{v}$, $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, and $\mathtt{v}_y \overset{*\circ}{\leftrightarrow}_V \mathtt{v}$. Here, we show only the case of $\mathtt{v}_y \overset{*+}{\leftrightarrow}_V \mathtt{v}$, in which case we let V' contain $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that $\widehat{\mathtt{x}}$ is removed from the sets $\mathtt{v}'.\mathtt{i}.\mathtt{pvars}$, $\mathtt{v}'.\mathtt{o}.\mathtt{pvars}$, $\mathtt{v}'.\mathtt{i}.\mathtt{reachfrom}$, $\mathtt{v}'.\mathtt{i}.\mathtt{reachto}$, $\mathtt{v}'.\mathtt{o}.\mathtt{reachfrom}$, and $\mathtt{v}'.\mathtt{o}.\mathtt{reachto}$.

The statement $\mathtt{x} := \mathtt{y}.\mathtt{next}[1]$ is handled rather similarly to the case $\mathtt{x} := \mathtt{y}$. Let us therefore describe the postcondition computation for statements of the form $\mathtt{x}.\mathtt{next}[1] := \mathtt{y}$. This is the most difficult statement, since it is a destructive update of the heap. It affects reachability relations for both $\mathtt{x}$ and $\mathtt{y}$. The postcondition computation makes a case analysis on how a fragment in V is related to some pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\widehat{\mathtt{x}} \in \mathtt{v}_x.\mathtt{i}.\mathtt{pvars}$, $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i}.\mathtt{pvars}$. Thus, for each pair of compatible fragments $\mathtt{v}_x$, $\mathtt{v}_y$ in V such that $\widehat{\mathtt{x}} \in \mathtt{v}_x.\mathtt{i}.\mathtt{pvars}$ and $\widehat{\mathtt{y}} \in \mathtt{v}_y.\mathtt{i}.\mathtt{pvars}$, it is first checked whether the statement may form a cycle in the heap. This may happen if $\mathtt{v}_y \overset{*}{\hookrightarrow}_V \mathtt{v}_x$, in which case the postcondition computation reports a potential cycle. Otherwise, V' consists of

1. the fragment $\mathtt{v}_{new}$, representing the new pair of neighbours formed by the statement, of form $\mathtt{v}_{new} = \langle \mathtt{i}, \mathtt{o}, \phi \rangle$, such that $\mathtt{v}_{new}.\mathtt{i}.\mathtt{tag} = \mathtt{v}_x.\mathtt{i}.\mathtt{tag}$ and $\mathtt{v}_{new}.\mathtt{o}.\mathtt{tag} = \mathtt{v}_y.\mathtt{i}.\mathtt{tag}$ except that $\mathtt{v}_{new}.\mathtt{o}.\mathtt{reachfrom} = \mathtt{v}_y.\mathtt{i}.\mathtt{reachfrom} \cup \mathtt{v}_x.\mathtt{i}.\mathtt{reachfrom}$ and $\mathtt{v}_{new}.\mathtt{i}.\mathtt{reachto} = \mathtt{v}_y.\mathtt{i}.\mathtt{reachto} \cup \mathtt{v}_x.\mathtt{i}.\mathtt{pvars}$; the constraint represent by $\mathtt{v}_{new}.\phi$ is obtained from the constraints represented by the data

abstractions of $\mathtt{v}_x.\mathtt{i}$ and $\mathtt{v}_y.\mathtt{i}$, as well as the possible transitive closure-relations between $\mathtt{v}_x$ and $\mathtt{v}_y$, some of which imply that the data fields of $\mathtt{v}_x$ and $\mathtt{v}_y$ are ordered, and

2. all possible fragments that can result from a transformation of some fragment $\mathtt{v} \in V$. This is done by an exhaustive case analysis on the possible relationships between $\mathtt{v}$, $\mathtt{v}_x$ and $\mathtt{v}_y$. Let us consider an interesting case, in which $\mathtt{v}_x \overset{*}{\hookrightarrow}_V \mathtt{v}$ and either $\mathtt{v} \overset{+}{\hookrightarrow}_V \mathtt{v}_y$ or $\mathtt{v}_y \overset{*+}{\leftrightarrow} \mathtt{v}$. In this case,
   - for each subset $\mathtt{regset}$ of the observer registers in $\mathtt{v.i.reachfrom} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$, and for each subset $\mathtt{regset}'$ of the set of observer registers in $\mathtt{v.o.reachfrom} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$, we let V' contain a fragment $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that $\mathtt{v}'\mathtt{.i.reachfrom} = (\mathtt{v.i.reachfrom} \setminus \mathtt{v_x.i.reachfrom}) \cup \mathtt{regset}$ and $\mathtt{v}'\mathtt{.o.reachfrom} = (\mathtt{v.o.reachfrom} \setminus \mathtt{v}_x\mathtt{.i.reachfrom}) \cup \mathtt{regset}'$. An intuitive explanation for the rule for $\mathtt{v}'\mathtt{.i.reachfrom}$ is that the global variables that can reach $\mathtt{v}_x\mathtt{.i}$ should clearly be removed from $\mathtt{v}'\mathtt{.i.reachfrom}$ since $\mathtt{v}_x \overset{*}{\hookrightarrow}_V \mathtt{v}'$ is false after the statement. However, for an observer register $\mathtt{x}_i$, an $\mathtt{x}_i$-cell can still reach $\mathtt{v}'\mathtt{.i}$, if there are two $\mathtt{x}_i$- cells, one which reaches $\mathtt{v}_x\mathtt{.i}$ and another which reaches $\mathtt{v}'\mathtt{.i}$; we cannot precisely determine for which $\mathtt{x}_i$ this may be the case, except that any such $\mathtt{x}_i$ must be in $\mathtt{v.i.reachfrom} \cap \mathtt{v}_x\mathtt{.i.reachfrom}$. The intuition for the rule for $\mathtt{v}'\mathtt{.o.reachfrom}$ is analogous.

Construction of higher-level fragments Based on the above construction of level-1 fragments, the set of higher-level fragments in V' is obtained as follows. For each higher level-fragment $\mathtt{v} \in V$, let $\mathtt{v}_1$ and $\mathtt{v}_2$ be level 1-fragments such that $\mathtt{v}_1\mathtt{.i.tag} = \mathtt{v.i.tag}$ and $\mathtt{v}_2\mathtt{.i.tag} = \mathtt{v.o.tag}$. For any fragments $\mathtt{v}'_1$ and $\mathtt{v}'_2$ that are derived from $\mathtt{v}_1$ and $\mathtt{v}_2$, respectively, V' contains a higher-level fragment $\mathtt{v}'$ which is the same as $\mathtt{v}$ except that (i) $\mathtt{v}'\mathtt{.i.pvars} = \mathtt{v}'_1\mathtt{.i.pvars}$ and $\mathtt{v}'\mathtt{.o.pvars} = \mathtt{v}'_2\mathtt{.i.pvars}$, (ii) $\mathtt{v}'\mathtt{.i.reachfrom} = \mathtt{v}'_1\mathtt{.i.reachfrom}$ and $\mathtt{v}'\mathtt{.o.reachfrom} = \mathtt{v}'_2\mathtt{.i.reachfrom}$, and (iii) $\mathtt{v}'\mathtt{.i.reachto} = \mathtt{v}'_1\mathtt{.i.reachto}$ and $\mathtt{v}'\mathtt{.o.reachto} = \mathtt{v}'_2\mathtt{.i.reachto}$. In addition, a statement of form $\mathtt{x.next}[\mathtt{k}] := \mathtt{y}$ for $\mathtt{k} \geq 2$ creates a new fragment. The formation of this fragment is simpler than for the statement $\mathtt{x.next}[1] := \mathtt{y}$, since reachability via $\mathtt{next}[1]$-pointers is preserved.

Symbolic Postcondition Computation for Interference Steps Here, the key step is the *intersection* operation, which takes two sets of fragments V_1 and V_2, and produces a set of joint fragments $V_{1,2}$, such that $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}_1,\mathtt{th}_2} V_{1,2}$ for any configuration such that $c_{\mathcal{S}} \models^{heap}_{\mathtt{th}_i} V_i$ for $i = 1, 2$ (here $\models^{heap}_{\mathtt{th}_1,\mathtt{th}_2}$ is defined in the natural way). This means that for each heap cell accessible to either $\mathtt{th}_1$ or $\mathtt{th}_2$, the set $V_{1,2}$ contains a fragment $\mathtt{v}$ with $\mathbb{c} \lhd^{c_{\mathcal{S}}}_{\{\mathtt{th}_1,\mathtt{th}_2\},\mathtt{k}} \mathtt{v}$ for each $\mathtt{k}$ which is at most the height of $\mathbb{c}$ (generalizing the notation $\lhd^{c_{\mathcal{S}}}_{\mathtt{th},\mathtt{k}}$ to several threads). Note that a joint fragment represents local pointer variables of both $\mathtt{th}_1$ and $\mathtt{th}_2$. In order to distinguish between local variables of $\mathtt{th}_1$ and $\mathtt{th}_2$, we use $\mathtt{x}[\mathtt{i}]$ to denote a local variable $\mathtt{x}$ of thread $\mathtt{th}_i$. Here, we describe the intersection operation for level-1 fragments. The intersection operation is analogous for higher-level fragments.

For a fragment $\mathtt{v}$, define $\mathtt{v.i.greachfrom}$ as the set of global variables in $\mathtt{v.i.reachfrom}$. Define $\mathtt{v.i.greachto}$, $\mathtt{v.o.greachfrom}$, $\mathtt{v.o.greachto}$, $\mathtt{v.i.gpvars}$, and $\mathtt{v.o.gpvars}$ analogously. Define $\mathtt{v.i.gtag}$ as the tuple $\langle \mathtt{v.i.dabs}, \mathtt{v.i.gpvars}, \mathtt{v.i.greachfrom}, \mathtt{v.i.greachto} \rangle$, and define $\mathtt{v.o.gtag}$ analogously. We must distinguish the following possibilities.

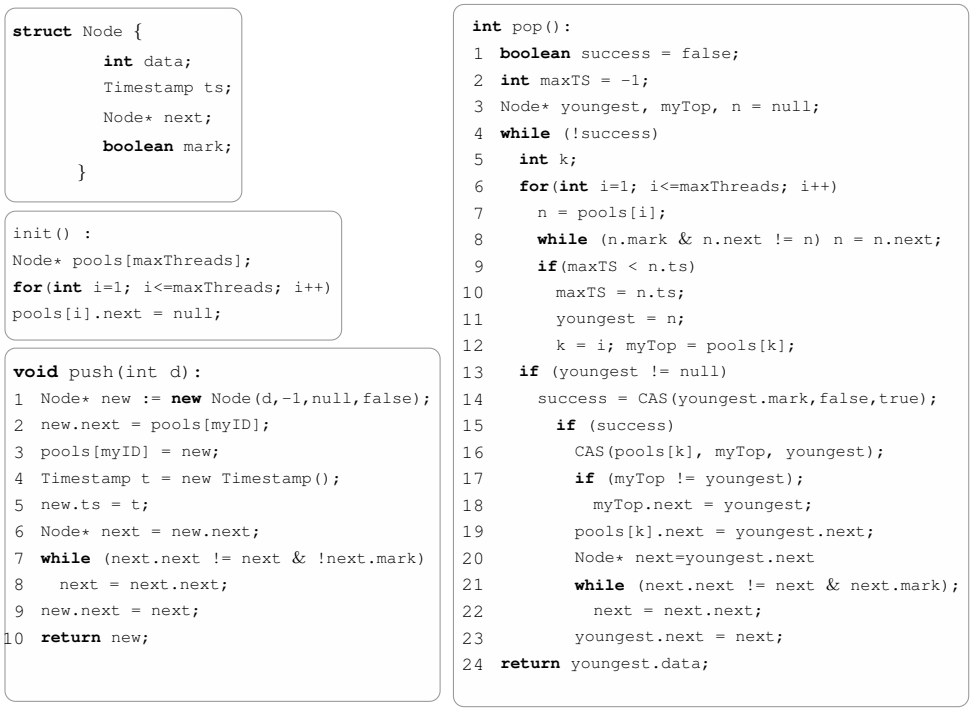
- If $\mathbb{c}$ is accessible to both $\mathtt{th}_1$ and $\mathtt{th}_2$, then there are fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$ and $\mathbb{c} \lhd^{cs}_{\mathtt{th}_2,1} \mathtt{v}_2$. This can happen only if $\mathtt{v}_1.\mathtt{i.gtag} = \mathtt{v}_2.\mathtt{i.gtag}$, and $\mathtt{v}_1.\mathtt{o.gtag} = \mathtt{v}_2.\mathtt{o.gtag}$, and $\mathtt{v}_1.\mathtt{i.private} = \mathtt{v}_2.\mathtt{i.private} = \mathtt{false}$. Thus, for any such pair of fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$, we let $V_{1,2}$ contain a fragment $\mathtt{v}_{12}$ which is identical to $\mathtt{v}_1$ except that
  - $\mathtt{v}_{12}.\mathtt{i.pvars} = \mathtt{v}_1.\mathtt{i.pvars} \cup \mathtt{v}_2.\mathtt{i.pvars}$,
  - $\mathtt{v}_{12}.\mathtt{o.pvars} = \mathtt{v}_1.\mathtt{o.pvars} \cup \mathtt{v}_2.\mathtt{o.pvars}$,
  - $\mathtt{v}_{12}.\mathtt{i.reachfrom} = \mathtt{v}_1.\mathtt{i.reachfrom} \cup \mathtt{v}_2.\mathtt{i.reachfrom}$, and
  - $\mathtt{v}_{12}.\mathtt{o.reachfrom} = \mathtt{v}_1.\mathtt{o.reachfrom} \cup \mathtt{v}_2.\mathtt{o.reachfrom}$.
- If $\mathbb{c}$ is accessible to $\mathtt{th}_1$, but not to $\mathtt{th}_2$, and $\mathbb{c}.\mathtt{next}[1]$ is accessible also to $\mathtt{th}_2$, then there are fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$ and $\mathbb{c}.\mathtt{next}[1] \lhd^{cs}_{\mathtt{th}_2,1} \mathtt{v}_2.\mathtt{o}$. This can happen only if $\mathtt{v}_1.\mathtt{i.greachfrom} = \emptyset$, and $\mathtt{v}_1.\mathtt{o.gtag} = \mathtt{v}_2.\mathtt{o.gtag}$, and $\mathtt{v}_1.\mathtt{o.private} = \mathtt{v}_2.\mathtt{o.private} = \mathtt{false}$. Thus, for any such pair of fragments $\mathtt{v}_1 \in V_1$ and $\mathtt{v}_2 \in V_2$, we let $V_{1,2}$ contain a fragment $\mathtt{v}'_1$ which is identical to $\mathtt{v}_1$ except that
  - $\mathtt{v}'_1.\mathtt{o.pvars} = \mathtt{v}_1.\mathtt{o.pvars} \cup \mathtt{v}_2.\mathtt{o.pvars}$, and
  - $\mathtt{v}'_1.\mathtt{o.reachfrom} = \mathtt{v}_1.\mathtt{o.reachfrom} \cup \mathtt{v}_2.\mathtt{o.reachfrom}$.
- If neither $\mathbb{c}$ nor $\mathbb{c}.\mathtt{next}[1]$ is accessible $\mathtt{th}_2$, then there is a fragment $\mathtt{v}_1 \in V_1$ such that $\mathbb{c} \lhd^{cs}_{\mathtt{th}_1,1} \mathtt{v}_1$. This can happen only if $\mathtt{v}_1.\mathtt{o.greachfrom} = \emptyset$, in which case we let $V_{1,2}$ contain the fragment $\mathtt{v}_1$.
- For each of the two last cases, there is also a symmetric case with the roles of $\mathtt{th}_1$ and $\mathtt{th}_2$ reversed.

5 Arrays of Singly-Linked Lists with Timestamps

In this section, we show how to apply fragment abstraction to concurrent programs that operate on a shared heap which represents an array of singly linked lists. We use this abstraction to provide the first automated verification of linearizability for the Timed-stamped stack and Timestamped queue algorithms of [12] as reported in Section 6.

Figure 8 shows a simplified version of the Timestamped Stack (TS stack) of [12], where we have omitted the check for emptiness in the `pop` method, and the optimization using `push`-`pop` elimination. These features are included in the full version of the algorithm, that we have verified automatically.

The algorithm uses an array of singly-linked lists (SLLs), one for each thread, accessed via the thread-indexed array `pools`[`maxThreads`] of pointers to the first cell of each list. The `init` method initializes each of these pointers to `null`. Each list cell contains a data value, a timestamp value, a `next` pointer, and a boolean flag `mark` which indicates whether the node is logically removed from the stack. Each thread pushes elements only to "its own" list, but can pop elements from any list.



```
struct Node {
        int data;
        Timestamp ts;
        Node* next;
        boolean mark;
      }
```

```
init() :
Node* pools[maxThreads];
for(int i=1; i<=maxThreads; i++)
pools[i].next = null;
```

```
void push(int d):
1  Node* new := new Node(d,-1,null,false);
2  new.next = pools[myID];
3  pools[myID] = new;
4  Timestamp t = new Timestamp();
5  new.ts = t;
6  Node* next = new.next;
7  while (next.next != next & !next.mark)
8    next = next.next;
9  new.next = next;
10 return new;
```

```
int pop():
 1 boolean success = false;
 2 int maxTS = -1;
 3 Node* youngest, myTop, n = null;
 4 while (!success)
 5   int k;
 6   for(int i=1; i<=maxThreads; i++)
 7     n = pools[i];
 8     while (n.mark & n.next != n) n = n.next;
 9     if(maxTS < n.ts)
10       maxTS = n.ts;
11       youngest = n;
12       k = i; myTop = pools[k];
13   if (youngest != null)
14     success = CAS(youngest.mark,false,true);
15       if (success)
16         CAS(pools[k], myTop, youngest);
17         if (myTop != youngest);
18           myTop.next = youngest;
19         pools[k].next = youngest.next;
20         Node* next=youngest.next
21         while (next.next != next & next.mark);
22           next = next.next;
23         youngest.next = next;
24 return youngest.data;
```

Fig. 8: Description of the Timestamped stack algorithm, with some simplifications.

A `push` method for inserting a data element `d` works as follows: first, a new cell with element `d` and minimal timestamp -1 is inserted at the beginning of the list indexed by the calling thread (line 1-3). After that, a new timestamp is created and assigned (via the variable `t`) to the `ts` field of the inserted cell (line 4-5). Finally, the method unlinks (i.e., physically removes) all cells that are reachable (through a sequence of `next` pointers) from the inserted cell and whose `mark` field is `true`; these cells are already logically removed. This is done by redirecting the `next` pointer of the inserted cell to the first cell with a `false` `mark` field, which is reachable from the inserted cell.

A `pop` method first traverses all lists, finding in each list the first cell whose `mark` field is `false` (line 8), and letting the variable `youngest` point to the most recent such cell (i.e., with the largest timestamp) (line 1-11). A compare-and-swap (CAS) is used to set the `mark` field of this youngest cell to `true`, thereby logically removing it. This procedure will restart if the CAS fails. After the youngest cell has been removed, the method will unlink all cells, whose `mark` field is `true`, that appear before (line 17-19) or after (line 20-23) the removed cell. Finally, the method returns the `data` value of the removed cell.

Fragment Abstraction In our verification, we establish that the TS stack algorithm of Figure 8 is correct in the sense that it is a linearizable implementation of a stack data structure. For stacks and queues, we specify linearizability by observers that synchro-

nize on call and return actions of methods, as shown by [7]; this is done without any user-supplied annotation, hence the verification is fully automated.

The verification is performed analogously as for skiplists, as described in Section 4. Here we show how fragment abstraction is used for arrays of singly-linked lists. Figure 9 shows an example heap state of TS stack. The heap consists of a set of singly linked lists (SLLs), each of which is accessed from a pointer in the array `pools[maxThreads]` in a configuration when it is accessed concurrently by three threads $\mathtt{th}_1$, $\mathtt{th}_2$, and $\mathtt{th}_3$. The heap consists of three SLLs accessed from the three pointers `pools[1]`, `pools[2]`, and `pools[3]` respectively. Each heap cell is shown with the values of its fields, using the layout shown to the right in Figure 9. In addition, each cell is labeled by the pointer variables that point to it. We use $\mathtt{lvar(i)}$ to denote the local variable `lvar` of thread $\mathtt{th}_\mathtt{i}$.

In the heap state of Figure 9, thread $\mathtt{th}_1$ is trying to push a new node with data value 4, pointed by its local variable `new`, having reached line 3. Thread $\mathtt{th}_3$ has just called the `push` method. Thread $\mathtt{th}_2$ has reached line 12 in the execution of the `pop` method, and has just assigned `youngest` to the first node in the list pointed to by `pools[3]` which is not logically removed (in this case it is the last node of that list). The observer has two registers $\mathtt{x}_1$ and $\mathtt{x}_2$, which are assigned the values 4 and 2, respectively.

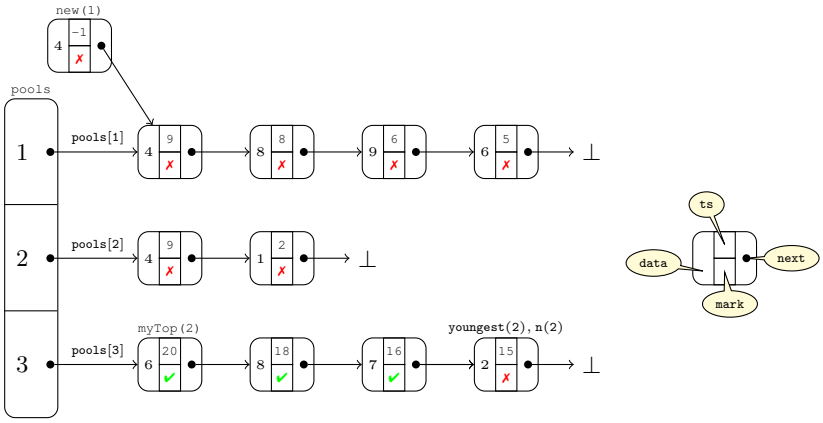


Fig. 9: A possible heap state of TS stack with three threads.

We verify the algorithm using a symbolic representation that is analogous to the one used for skiplists. There are two main differences.

- Since the array `pools` is global, all threads can reach all lists in the heap (the only cells that cannot be reached by all threads are new cells that are not yet inserted).
- We therefore represent the view of a thread by a thread-dependent abstraction of thread indices, which index the array `pools`. In the view of a thread, the index of the list where it is currently active is abstracted to `me`, and all other indices are abstracted to `ot`. The currently active index is taken to be the thread index for a

thread performing a `push`, the value of `i` for a thread executing in the **for** loop of `pop`, and the value of `k` after that loop.

In the definition of tags, the only global variables that can occur in the fields `reachfrom` and `reachto` are therefore `pools[me]` and `pools[other]`. The data abstraction represents (i) for each cell, the set of observer registers, whose values are equal to the `datafield`, (ii) for each timestamp and observer register x_i, the possible orderings between this timestamp and the timestamp of an x_i-cell.

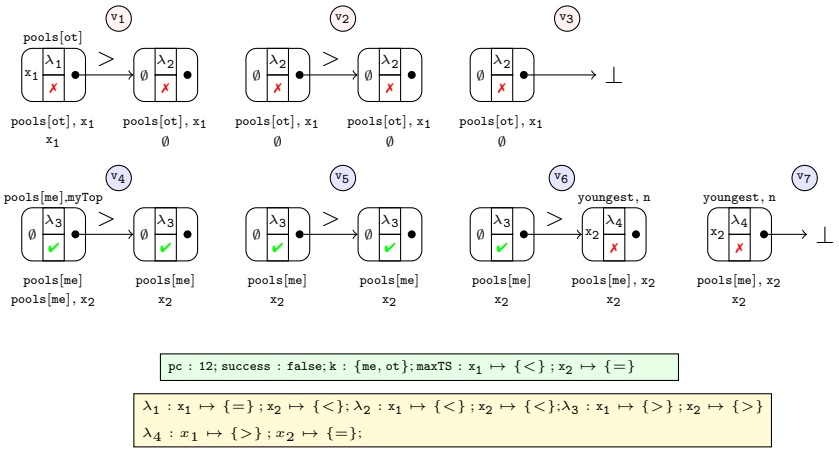


Fig. 10: Fragment abstraction

Figure 10 shows a set of fragments that is satisfied wrp. to $\mathtt{th}_2$ by the configuration in Figure 9. There are 7 fragments, named $\mathtt{v}_1, \ldots, \mathtt{v}_7$. Consider the tag which occurs in fragment $\mathtt{v}_7$. This tag is an abstraction of the bottom-rightmost heap cell in Figure 9, The different non-pointer fields are represented as follows.

- The `data` field of the tag (to the left) abstracts the data value 2 to the set of observer registers with that value: in this case $\mathtt{x}_2$.
- The `ts` field (at the top) abstracts the timer value 15 to the possible relations with `ts`-fields of heap cells with the same data value as each observer registers. Recall that observer registers $\mathtt{x}_1$ and $\mathtt{x}_2$ have values 4 and 2, respectively. There are three heap cells with `data` field value 4, all with a `ts` value less than 15. There is one heap cell with `data` field value 2, having `ts` value 15. Consequently, the abstraction of the `ts` field maps $\mathtt{x}_1$ to $\{>\}$ and $\mathtt{x}_2$ to $\{=\}$: this is the mapping λ_4 in Figure 10.
- The `mark` field assumes values from a small finite domain and is represented precisely as in concrete heap cells.

Symbolic Postcondition Computation The symbolic postcondition computation is similar to that for skiplists. Main differences are as follows.

- Whenever a thread performing `pop` moves from one iteration of the `for` loop to the next, the abstraction must consider to swap between the abstractions `me` and `ot`.

– In interference steps, we must consider that the abstraction `me` for the interfering thread may have to be changed into `ot`. Furthermore, the abstractions `me` for two `push` methods cannot coincide, since each thread pushes only to its own list.

6 Experimental Results

Based on our framework, we have implemented a tool in OCaml, and used it for verifying various kinds of concurrent data structures implementation of stacks, priority queues, queues and sets. All of them are based on heap structures. There are three types of heap structures we consider in our experiments.

Algorithms	**Time (s)**		**Algorithms**	**Time (s)**	
	a	b		a	b
Treiber stack [39]	18	0.18	O'Hearn set [32]	88	12
MS lock-free queue [31]	22	21	HM lock-free set [22]	120	462
DGLM queue [14]	16	16	Harris lock-free set [17]	950	1512
Vechev-CAS set [44]	86	24	Unordered set [48]	1230	2301
Vechev-DCAS set [44]	16	16	TS stack [12]	176	
Michael lock-free set [29]	178	110	TS queue [12]	101	
Pessimistic set [22]	30	1.51	Lock-free skiplist [22]	1992	
Optimistic set [22]	25	60	Lock-based skiplist [21]	500	
Lazy set [20]	34	289	Priority queue skiplist 1 [27]	1320	
			Priority queue skiplist 2 [26]	599	

Fig. 11: Times for verifying concurrent data structure implementations. Column **a** shows the verification times for our tool based on fragment abstraction. Column **b** shows the verification times for the tool for SLLs in our previous work [3]

Singly-linked list benchmarks: These benchmarks include stacks, queues and sets algorithms which are the well-known in the literature. The challenge is that in some set implementation, the linearization points are not fixed, they depended on the future of each execution. The sets with non fixed linearization points are the lazy set [20], lock-free sets of *HM* [22], *Harris* [17], *Michael* [29], and unordered set of [48]. By using observers and controllers in our previous work [3]. Our approach is simple and strong enough to verify these singly-linked list benchmarks.

Skiplist benchmarks: We consider four skiplist algorithms including the lock-based skiplist set [31], the lock-free skiplist set which is described in section 2 [22], and two skiplist-based priority queues [26], [27]. One challenge for verifying these algorithms is to deal with unbounded number of levels. In addition, in the lock-free skiplist [22] and priority queue [26], the skiplist shape is not well formed, meaning that each higher level list need not be a sub-list of lower level lists. These algorithms have not been automatically verified in previous work. By applying our fragment abstraction, to the best of our knowledge, we provide first framework which can automatically verify these concurrent skiplists algorithms.

Arrays of singly-linked list benchmarks: We consider two challenging timestamp algorithms in [12]. There are two challenges when verifying these algorithm. The first challenge is how to deal with an unbounded number of SLLs, and the second challenge is that the linearization points of the algorithms are not fixed, but depend on the future of each execution. By combining our fragment abstraction with the observers for stacks and queues in [7], we are able to verify these two algorithms automatically. The observers are crucial for achieving automation, since they enforce the weakest possible ordering constraints that are necessary for proving linearizability, thereby making it possible to use a less precise abstraction.

Running Times. The experiments were performed on a desktop 2.8 GHz processor with 8GB memory. The results are presented in Fig. 11, where running times are given in seconds. Column a shows the verification times of our tool, whereas column b shows the verification times for algorithms based on SLLs, using the technique in our previous work [3]. In our experiments, we run the tool together with an observer in [1], [7] and controllers in [3] to verify linearizability of the algorithms. All experiments start from the initial heap, and end either when the analysis reaches a fixed point or when a violation of safety properties or linearizability is detected. As can be seen from the table, the verification times vary in the different examples. This is due to the types of shapes that are produced during the analysis. For instance, skiplist algorithms have much longer verification times. This is due to the number of pointer variables and their complicated shapes. In contrast, other algorithms produce simple shape patterns and hence they have shorter verification times.

Error Detection In addition to establishing correctness of the original versions of the benchmark algorithms, we tested our tool with intentionally inserted bugs. For example, we omitted setting time statement in line 5 of the `push` method in the TS stack algorithm, or we omitted the `CAS` statements in lock-free algorithms. The tool, as expected, successfully detected and reported the bugs.

7 Conclusions

We have presented a novel shape abstraction, called fragment abstraction, for automatic verification of concurrent data structure implementations that operate on different forms of dynamically allocated heap structures, including singly-linked lists, skiplists, and arrays of singly-linked lists. Our approach is the first framework that can automatically verify concurrent data structure implementations that employ skiplists and arrays of singly linked lists, at the same time as handling an unbounded number of concurrent threads, an unbounded domain of data values (including timestamps), and an unbounded shared heap. We showed fragment abstraction allows to combine local and global reachability information to allow verification of the functional behavior of a collection of threads.

As future work, we intend to investigate whether fragment abstraction can be applied also to other heap structures, such as concurrent binary search trees.

References

1. Abdulla, P.A., Haziza, F., Holík, L., et al.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS. LNCS, vol. 7795, pp. 324–338 (2013)
2. Abdulla, P.A., Holík, L., Jonsson, B., Trinh, C.Q., et al.: Verification of heap manipulating programs with ordered data by extended forest automata. Acta Inf. 53(4), 357–385 (2016)
3. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Automated verification of linearization policies. In: SAS. LNCS, vol. 9837, pp. 61–83. Springer (2016)
4. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV'07. LNCS, vol. 4590, pp. 477–490 (2007)
5. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: CAV'08. LNCS, vol. 5123, pp. 399–413 (2008)
6. Bingham, J., Rakamaric, Z.: A logic and decision procedure for predicate abstraction of heap-manipulating programs. In: VMCAI. LNCS, vol. 3855, pp. 207–221. Springer (2006)
7. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In: ICALP. LNCS, vol. 9135, pp. 95–107. Springer (2015)
8. Bouajjani, A., Emmi, M., Enea, C., Mutluergil, S.O.: Proving linearizability using forward simulations. In: CAV. LNCS, vol. 10427, pp. 542–563. Springer (2017)
9. Chakraborty, S., Henzinger, T.A., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. Logical Methods in Computer Science 11(1) (2015)
10. Chang, B.Y., Rival, X., Necula, G.: Shape Analysis with Structural Invariant Checkers. In: Proc. of SAS'07. LNCS, vol. 4634. Springer (2007)
11. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: CAV. LNCS, vol. 4144, pp. 475–488 (2006)
12. Dodds, M., Haas, A., Kirsch, C.: A scalable, correct time-stamped stack. In: POPL. pp. 233–246. ACM (2015)
13. Doherty, S., Detlefs, D., Groves, L., Flood, C., et al.: DCAS is not a silver bullet for nonblocking algorithm design. In: SPAA'04. pp. 216–224. ACM (2004)
14. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE'04. LNCS, vol. 3235, pp. 97–114. Springer (2004)
15. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: SAS. LNCS, vol. 7935, pp. 215–237. Springer (2013)
16. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC'04. pp. 50–59. ACM (2004)
17. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. LNCS, vol. 2180, pp. 300–314. Springer (2001)
18. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: VMCAI. Lecture Notes in Computer Science, vol. 9583, pp. 393–412. Springer (2016)
19. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS 266, 93–107 (2010)
20. Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W.N.S., Shavit, N.: A lazy concurrent list-based set algorithm. In: OPODIS. LNCS, vol. 3974, pp. 3–16. Springer (2005)
21. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skip-list algorithm. In: SIROCCO. LNCS, vol. 4474, pp. 124–138. Springer (2007)
22. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
23. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: CAV. LNCS, vol. 8044, pp. 740–755. Springer (2013)
24. Khyzha, A., Dodds, M., Gotsman, A., Parkinson, M.: Proving linearizability using partial orders. In: ESOP. LNCS, vol. 10201, pp. 639–667. Springer (2017)

25. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. pp. 459–470. ACM (2013)
26. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: OPODIS. LNCS, vol. 8304, pp. 206–220. Springer (2013)
27. Lotan, I., Shavit, N.: Skiplist-based concurrent priority queues. In: IPDPS. pp. 263–268. IEEE (2000)
28. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, S.: Predicate abstraction and canonical abstraction for singly-linked lists. In: VMCAI. pp. 181–198. LNCS (2005)
29. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82 (2002)
30. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Tech. Rep. TR599, University of Rochester, Rochester, NY, USA (1995)
31. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC. pp. 267–275. ACM (1996)
32. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC. pp. 85–94 (2010)
33. Sagiv, S., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. on Programming Languages and Systems 24(3), 217–298 (2002)
34. Sánchez, A., Sánchez, C.: Formal verification of skiplists with arbitrary many levels. In: ATVA. LNCS, vol. 8837, pp. 314–329. Springer (2014)
35. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. 15(4), 31:1–37 (2014)
36. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: APLAS. LNCS, vol. 5904, pp. 30–46. Springer (2009)
37. Singh, V., Neamtiu, I., Gupta, R.: Proving concurrent data structures linearizable. In: ISSRE. pp. 230–240. IEEE (2016)
38. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (May 2005)
39. Treiber, R.: Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Res. Ctr. (1986)
40. Turon, A.J., Thamsborg, J., Ahmed, A., Birkedal, L., Dreyer, D.: Logical relations for fine-grained concurrency. In: POPL '13. pp. 343–356. ACM (2013)
41. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2008)
42. Vafeiadis, V.: Automatically proving linearizability. In: CAV. LNCS, vol. 6174, pp. 450–464 (2010)
43. Černý, P., Radhakrishna, A., Zufferey, D., et al.: Model checking of linearizability of concurrent list implementations. In: CAV. LNCS, vol. 6174, pp. 465–479 (2010)
44. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135. ACM (2008)
45. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: SPIN. LNCS, vol. 5578, pp. 261–278. Springer (2009)
46. Wachter, B., Westphal, B.: The spotlight principle. In: VMCAI. LNCS, vol. 4349, pp. 182–198. Springer (2007)
47. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. In: CAV'08. LNCS, vol. 5123. Springer (2008)
48. Zhang, K., Zhao, Y., Yang, Y., Liu, Y., Spear, M.F.: Practical non-blocking unordered lists. In: DISC. LNCS, vol. 8205, pp. 239–253. Springer (2013)