Model Checking Unbounded Concurrent Lists

Divjyot Sethi[1], Muralidhar Talupur[2], and
Sharad Malik[1]

[1] Princeton University
[2] Strategic CAD Labs,
Intel Corporation

Abstract. We present a method for model checking list-based concurrent data structures. These data structures, increasingly available in libraries such as Intel Thread Building blocks and Java.util.concurrent (JSR), are notorious for being error prone. This stems from the usage of sophisticated synchronization techniques in their implementation for high efficiency. This efficiency comes at the cost of increasing the number of possible thread interleavings during execution, thus making them hard to verify. Consequently, the verification of concurrent data structures has been of interest to the formal methods community.
Concurrent data structures are unbounded in two dimensions: the list size is unbounded and an unbounded number of threads access them. Thus, their model checking requires abstraction to a model bounded in both the dimensions. In previous work, we showed how the unbounded threads can be model checked by using the CMP (CoMPositional) method. The method abstracted the unbounded threads by keeping one thread as is and abstracting all the other threads to a single environment thread. Next, this abstraction was iteratively refined by the user in order to prove correctness. However, in that work we assumed that the number of list elements were bounded to a fixed value. In practice this fixed value was small; model checking could only complete for small sized lists.
In this work, we overcome this limitation and model check the unbounded list as well. Our method has the same flavor as the CMP method in the thread dimension, but differs significantly in the list dimension. In the list dimension, our method constructs an abstraction of the unbounded list by keeping the nodes pointed to by the pointers in the model as is (referred to as *concrete nodes*), and abstracting away chains of all the nodes which are not pointed to by any pointers to *abstract nodes*. Since the accesses to the abstract nodes return non-deterministic field values, the abstraction may have to be refined to constrain these values. This is accomplished by the iterative addition of lemmas by the user. We show the soundness of our method and establish its utility by model checking challenging concurrent data structure examples.

1 Introduction

We present a method for model checking list-based concurrent data structures. These data structures are highly efficient concurrent list-based implementations of popular data structures such as sets, queues etc. and are increasingly available in libraries such as Intel Thread Building Blocks and Java.util.concurrent. These list-based implementations utilize sophisticated synchronization techniques, such as fine-grained locking or

lock free synchronization, to achieve high efficiency. Due to the complex synchronization used, these data structures are notorious for being highly error prone, as exemplified by bugs in published algorithms [13]. Consequently, verification of these data structures has been of interest to the verification community [1–4, 6, 20–22, 24, 25].

Linearizability [9] is the widely accepted correctness criterion for concurrent data structures. Intuitively, Linearizability implies that the execution of every access method of the concurrent data structure appears to occur atomically at some point – the *linearization point* – between the invocation and the response of the method.

In previous work [18], we showed how to model check Linearizability for concurrent data structures. Concurrent list-based data structures are unbounded in two dimensions – they have an unbounded number of list nodes and an unbounded number of threads accessing the list items. In our work, we verified these data structures for an unbounded number of threads. This was accomplished by using the CMP (CoMPositional) method [5]. The CMP method is used to verify symmetric parameterized systems of the form $P(N)$, with N identical threads $1..N$. The properties verified are candidate invariants of the form $\forall i \in [1..N].\Phi(i)$, where $\Phi(i)$ is a propositional logic formula on the variables of thread i and shared variables.

The CMP method exploits symmetry and locality; i.e., it assumes that the violation occurs at a particular thread, say thread 1. Consequently, the CMP method constructs an abstract model which consists of one thread from the original system (say thread 1 since the system is symmetric) and an environment thread (named *Other*) that over-approximates the remaining threads. Then, if the property $\Phi(1)$ holds on thread 1 in the abstract model, $\forall i \in [1..N].\Phi(i)$ holds for $P(N)$ by symmetry. The verification of the abstract model is done by using a model checker. The refinement of the abstract model is done in a loop, referred to as the CMP loop. In the loop, if the abstract model is falsified by the model checker, the model is refined by the user by supplying non-interference lemmas, in order to constrain the *other* thread [18].

The key advantage of using the CMP method is that the user-added lemmas also get verified. Thus, the CMP method is sound in the sense that if the CMP loop converges and the property is verified, all user-added lemmas along with the property under check hold. Another advantage of the CMP method is that it uses a model checker as a proof assistant. Thus, the added lemmas together with the property under check need not add up to be inductive, unlike most theorem proving based approaches [19]. These features in practice have been instrumental in making the CMP method useful in verifying complex cache coherence protocols [15, 19].

Since the CMP method does not handle unbounded list size, in our initial work [18], we had assumed the list size to be bounded. Further, in practice, we were only able to scale to a small number of list nodes. This, we believe was primarily due to the large number of interleavings in the execution of concurrent data structures and is consistent with the limited success of other model checking based efforts for concurrent data structures [4, 22, 24, 25]. The limited scalability in bounding the number of list nodes motivated us to study possible extensions of the CMP method approach in order to model check Linearizability for data structures with an unbounded list size as well.

1.1 Challenges in extending the CMP method to the list dimension

The list dimension has some key differences from the thread dimension which make the extension of the CMP method to the list dimension challenging. First, unlike the thread dimension, the list dimension lacks symmetry: the heap elements are connected asymmetrically depending on the heap shape. Second, in the list dimension, while checking the correctness can be localized to the few nodes that are being updated (additions and deletions), these localized nodes will change as the list is traversed. This dynamic nature of the nodes of interest precludes consideration of one or a small set of fixed nodes for the abstract model.

1.2 Extending the CMP Method to list dimension

In this work we extend the CMP method to the list dimension and show how abstraction and refinement can be done in this dimension.

Abstraction Intuition: The abstraction in the list dimension has been used earlier and is straightforward. It proceeds by retaining the nodes pointed to by the pointers in the program as is. These retained nodes are referred to as *concrete nodes*. Next, all nodes which are not pointed to by any pointers are abstracted. This is done by replacing all chains of nodes, such that no node in the chain is pointed to by any pointer, by *abstract nodes*. Fig 1a shows the intuition behind replacing a chain of concrete nodes by an abstract node. In the figure, nodes 2 and 3 are replaced by the abstract node (shown by a rectangle) on abstraction.

Observe that as the program state evolves, the pointers move. This moves the position of the concrete nodes as well. This movement of concrete nodes happens when the abstract nodes are accessed by pointers during transitions. Intuitively, access to an abstract node can be understood as a non-deterministic access to some concrete node which is a part of the chain of concrete nodes represented by the abstract node.

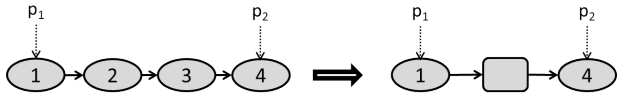
As an example in Fig 1b, when pointer p_3 accesses the abstract node, the abstract node is split into a concrete node with an abstract node on each side. The concrete node is assigned a non-deterministic value v. The pointer p_3 then points to this newly created concrete node. Since this newly created node v has a non-deterministic value, it may lead to a violation of the property under check. Thus, a refinement step may be required to constrain the value taken by v.

Refinement Intuition: Refinement is done by constraining the value v. This is done by specifying list lemmas (invariants on the heap) which v should not violate. As an example, a list lemma could state that the list elements are ordered (also referred to as *ordering* invariant). Then, the splitting may be done by assuming the invariant that the list nodes are ordered; and thus $v \in (1, 4)$.

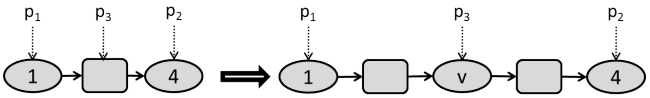
1.3 Framework description:

Fig 2 shows the extended CMP method loop with the extensions circled. As shown in the figure, the loop proceeds by first abstracting the threads as in the prior CMP method approach. Next, the unbounded list is abstracted. [3] On abstraction, in case the model

[3] In general, the abstractions are commutative. But, we present them in the order of the thread abstraction first and then the list abstraction.



(a) Abstraction: nodes 2 and 3 are replaced by an abstract node (represented by a rectangle)



(b) Splitting: abstract node is split into multiple nodes.

Fig. 1: Abstract node creation and splitting: the concrete nodes are shown as circles with indicated values and the abstract nodes are represented as rectangles.

is falsified by the model checker, the user inspects the counter-example. If the counter-example is a valid counter-example, a real bug has been found and the loop terminates. On the other hand, if the counter-example is spurious, the user refines the model in either the thread or list dimension.

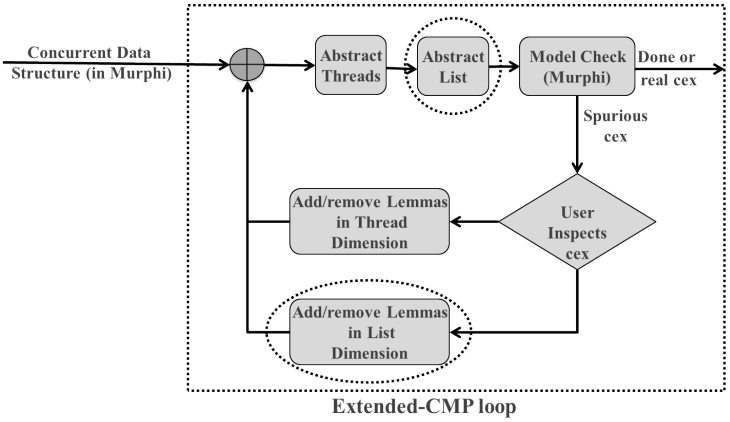


Fig. 2: Extended CMP Method: the circled steps indicate the key extensions in this work

1.4 Key Contributions:

Our extension preserves the advantages of the CMP method: in case the extended CMP loop converges, both, the property under check and the added invariants to constrain the list are proven to be correct. Thus the extended CMP loop is sound. Further, as in the CMP method, the added invariants need not add up to be inductive.

We make the following contributions in this work:

- We extend the CMP method to the list dimension and provide a syntactic abstraction of the unbounded list (Section 3) and mechanisms for refinement in the list dimension (Section 4).
- We show the soundness of the extended CMP method and show how the added list lemmas are also proven correct if the extended loop converges. (Section 4).
- We establish the utility of our method by model checking Linearizability of the *Fine-grained* [8] and *Optimistic* [8] data structures. (Section 5)

Key Limitations: (1) The user has to manually come up with lemmas to refine the model. However due to the small length of concurrent data structure implementations (most implementations fit in one page [8]), this has not been a problem in practice. (2) The abstraction used for verifying certain key properties of the *Optimistic* algorithm did not scale well. This is because of a large number of interleavings.

1.5 Related Work

Concurrent data structures have been proposed as a promising approach to harness the power of multi-core processors [8]. Given their importance and the challenges associated with proving them right, verification of concurrent data structures is garnering much recent attention [1–4, 6, 20–22, 24, 25].

In [21], verification of concurrent data structures was done using a mechanical proof assistant and the interference between concurrent threads was specified using rely and guarantee conditions. These approaches allow both arbitrary number of accessing threads and elements but they are manual effort intensive.

Among model checking approaches, Vechev et al. [22] describe their experience with verifying Linearizability using the SPIN model checker. Similarly, the work by Zhang et al. [24, 25] and Liu et al. [11] also performs model checking to verify concurrent data structures. Both these approaches use a refinement based proof approach and scale to not more than a small number of threads and list nodes. A different approach is taken by Alur et al. [4]: they treat a list as a string and the thread as an automaton and then derive conditions on the automaton to prove decidability of Linearizability. The key focus of their work is on decidability instead of scalability; while they handle unbounded list size, they do not scale to more than 2-3 threads. Noll et al. [14] handle unbounded lists and unbounded number of threads. Their list abstraction is similar to ours. In the thread dimension, instead of fully throwing away threads 2..N, they do a counter abstraction which results in increased state. Further, their refinement approach in both list and thread dimensions results in more added state, thus limiting scalability (they only apply their method to a small example).

Among analysis based approaches for verifying Linearizability of concurrent data structures, the RGSep based approach [20] works by combining separation logic with rely-guarantee reasoning into a logic called RGSep. Their approach is automatic for a subset of RGSep. While automatic, the designer still needs to specify concurrent actions which model the interference, just like in rely-guarantee reasoning, for proving assertions. Further, the designer also has to have an understanding of RGSep. In contrast,

our method requires the user to specify lemmas, which are standard Boolean formulas. Further, these lemmas are checked for correctness as well: any false lemmas will be weeded out. Next, shape analysis based approaches, like based on the tool TVLA [1] and thread modular analysis [7], are able to verify unbounded concurrent data structures. The primary focus of these approaches is to lift the heap abstractions for single threaded programs to concurrent programs. In [23], multiple threads in the same local state are abstracted into one abstract configuration and mechanisms for refinement through addition of more predicates are provided. In contrast to these approaches, our method focusses on mechanisms for efficient abstraction and refinement without the addition of extra state, to enable efficient model checking. Further, the user-supplied lemmas for refinement are also checked for correctness.

Finally, while the CMP (CoMPositional) method [5] is a highly successful parameterized verification technique, numerous work has been done in the area of parameterized verification, for e.g. [10, 16, 17]. While some of these classical methods are potentially applicable to concurrent data structures as well, given the success of the CMP method in verifying industrial cache coherence protocols [15], we believe that our method, which has a similar flavor to the CMP method, provides a scalable and effective alternative approach.

2 Modelling Concurrent Data Structures

2.1 Preliminaries

We model a concurrent list-based data structure as a program $P(M, N)$ with M heap nodes and N *identical* threads with ids $1..N$, where M and N are arbitrary but fixed. The set of threads is denoted by $Threads$ and the set of heap nodes is denoted by $List$.

Each node n of the heap consists of the fields key, $next$, $lock$ and $\{l_1, l_2, \ldots\}$, a finite set of local fields with values from a finite domain. The key field is defined on a generic data domain D which has comparison ($<$) and equality ($=$) operations. The $next$ field is used to indicate the adjacent node. If $n_j = n_i.next$, we say that n_j is a *successor* of n_i (or next to n_i), and that n_i is the *predecessor* of n_j. Finally, the $lock$ field takes values from $0..N$, where 0 represents unlocked and value i represents locked by thread i.

The heap nodes are pointed to by *global pointers* and by threads. Each thread i consists of a finite number of local state variables (with finite domain) for program execution and a finite number of *pointer variables* which access the heap nodes. We assume that the local variables of all threads (both local state and pointer variables) are stored in arrays ranging over thread ids [1..N]. Thus, a local variable v of thread i is written as $v[i]$. Similarly, $p[i]$ represents a local pointer of thread i.

Verification of concurrent data structures is done by checking for refinement against a specification set [21]. The specification set, denoted by S, is a sequential specification of the set implemented by the concurrent data structure implementation. The set S is essentially a set of key values from the domain D.

Transitions: Following the approach of [4], we model each statement of thread i as a transition of the form (l_i, G, Act, Ins, l'_i), where l_i and l'_i are initial and final states

of thread i for the transition, G is the guard, Act is an action on the heap and Ins is the instrumented action on the set S. The guards G are Boolean expressions constructed on the global pointers, thread local state variables and their values, and thread local pointer variables.

The thread actions are defined as follows:

Heap Traversal: $p_a[i] := p_b[i]$, $p_a[i] := p_b[i].next$,
Heap shape update: $p_a[i].next := p_b[i]$, and
Heap node update: $p[i].field = val$, where val is a value of appropriate type. [4]

Finally, the instrumented action Ins can either be (1) a nil action or, (2) an addition of $e \in D$ to S, denoted by $seqAdd(e)$ or, (3) a removal of an element $e \in D$ from S, denoted by $seqRemove(e)$ or, (4) a check for containment of an element $e \in D$ in S, denoted by $seqContains(e)$.

Definitions: We define predicate $\mathcal{R}_{p1}(p)$ to indicate that the node pointed to by p is reachable from that pointed to by $p1$. $\widehat{\mathcal{R}}_p$ refers to the set of all the nodes reachable from p. This definition extends in an obvious way to a set of pointers. Next, we call a node a *referred* node if it is pointed to by a thread pointer or a global pointer. We write $Ref_i(n)$ to indicate that some pointer in thread i points to node n. Further, we use $Ref(n)$ to indicate that the node n is pointed to by some pointer (thread local or global pointer) in the system. Finally, we define $isMuInt(n_i, n_j)$ to indicate that no node from the list segment n_i to n_j is referred to by any thread or global pointer in the program and further, the predecessor of n_i and successor of n_j are both referred nodes. Such a list segment is also referred to as a *maximally uninterrupted* list segment.

Property: In this paper we focus on verifying invariants specified on the above program. These invariants are of the form $\forall i : \phi(i)$, where $\phi(i)$ is an invariant involving the local variables of thread i and global variables quantified on the heap.

2.2 Running Example

We present the ideas in this paper through a *Fine-grained* list-based set ([8]) implementation. This implementation uses a linked list to implement a standard set. The methods in the implementation are the standard *Add*, *Remove* and *Contains* methods. The linked list consists of nodes with fields: 1) a *key* field holding values as integers, 2) a *next* pointer for accessing the next node in the list, and 3) a *lock* field representing whether that node is currently locked. In addition, there are two special (sentinel) nodes, the node *Head* and the node *Tail*, that can neither be added nor be removed. These nodes are pointed to by global pointers H and T respectively.

The concurrent implementation consists of potentially an unbounded number of threads. These threads are assumed to be symmetric. [5] Instead of locking the entire list, each method of the *Fine-grained* data structure traverses the list by using *hand-in-hand* locking. Fig 3a shows the implementation of the *Remove* function of the *Fine-grained* data structure. The *Remove* method uses hand-in-hand locking during traversal: $p_0[i]$

[4] This includes synchronization mechanisms; i.e., updates to the $lock$ field.

[5] While the threads are symmetric, they can execute different methods (such as *Add* or *Remove* or *Contains*) by non-deterministically calling any of these methods.

is unlocked (line 6), pointed to successor $p_1[i]$ (line 7), $p_1[i]$ is advanced (line 8) and locked again (line 9).

The concurrent list-based implementation implements a sequential set specification, denoted by S. The methods of the specification set are denoted by $seqAdd$, $seqRemove$ and $seqContains$. These methods have standard sequential set semantics.

Murphi Encoding: We show the encoding of the *Remove* method in the Murphi language for model checking. Our choice of Murphi was based on the powerful model checker which comes along with Murphi, and the legacy implementation of both, our abstraction tool (Abster [19]) and *Fine-grained* and *Optimistic* data structures.

Murphi uses a standard guard-action based syntax: the applications are written as a collection of rules of the form $\rho \Rightarrow a$, where ρ is the guard and a is the action. Fig 3b shows the guard-action based encoding of a few statements of the method. [6] Observe that the variable $pc[i]$ represents the program counter and is used to enforce a sequential execution of the rules. This is done in order to simulate the sequential execution of statements within a thread, since we assume a sequentially consistent memory model. This guard-action based encoding is useful in presenting the ideas in this paper.

Remove (key)
1: $p_0[i] := H$;
2: $p_0[i].lock()$;
3: $p_1[i] := p_0[i].next$;
4: $p_1[i].lock()$;
5: while ($p_1[i].key < key$)
6: $p_0[i].unlock()$;
7: $p_0[i] := p_1[i]$;
8: $p_1[i] := p_0[i].next$;
9: $p_1[i].lock()$;
10: if $p_1[i].key = key$ then
11: $p_2[i] := p_1[i].next$;
12: $p_0[i].next := p_2[i]$;
[*SeqRemove(key)]
13: $result := true$;
14: else
15: $result := false$;
[*SeqRemove(key)]
16: $p_0[i].unlock()$;
17: $p_1[i].unlock()$;

(a) Pseudo-code for linked list-based *Fine-grained* set algorithm. The linearization points are marked with a $*$.

$\forall i \in Threads : (pc[i] = 1) \rightarrow \{$
$p_0[i] := H;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 3) \rightarrow \{$
$p_1[i] := p_0[i].next;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 12) \rightarrow \{$
$p_0[i].next := p_2[i];$
$pc[i] + +;$
[*SeqRemove(key)] $\}$

(b) Murphi based guard-action pairs for statements with line numbers 1,3 and 12

Fig. 3: *Remove* function and Murphi encoding.

[6] The complete guard-action based encoding of the *Remove* method is provided in Appendix A.1.

Example State: Fig 4 shows an example of a state in the execution of the list-based implementation. In this state, the linked list is accessed by 3 threads, with ids 1, 2 and 3. Further, the value of the specification set S is also shown in the figure. Observe that $ConcSet$ and S match, where $ConcSet$ is the set of key values of all the nodes reachable from H. Next, observe that $Ref(1)$ is false but $Ref(0)$ is true. Further, $isMuInt(1,1)$, $isMuInt(9,9)$ are true and $isMuInt(n_i, n_j)$ is false for all other pairs of n_i and n_j, as all the other nodes are referred nodes.

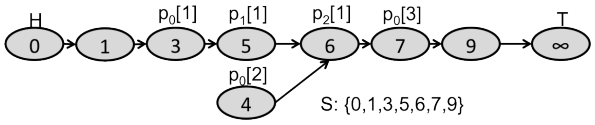


Fig. 4: Heap accessed by pointers of threads 1,2 and 3. $p_j[i]$ denotes the j^{th} pointer of thread i.

Property under verification *Linearizability* [9] is a key property of interest in proving the correctness of concurrent data structures. Intuitively, Linearizability implies that the execution of every access method for the concurrent data structure appears to occur at some point between the invocation and the response of the method. This point is referred to as the *linearization point*.

In this work, we verify Linearizability by a refinement based approach, as described in [21]. The refinement is proven against the specification set S. This is done by matching the results of the call to S against the return value of the implementation method. Calls to the S are inserted at the linearization point.

As an example, for the *Remove* method shown in Fig 3a, the linearization point for *Remove* in case the call is successful is marked with [**SeqRemove(key)*] on Line 3. Similarly, [**SeqRemove(key)*] on Line 8 denotes the linearization point in the failing case. If both, the concurrent methods and the embedded specification methods return the same value, the concurrent data structure is Linearizable. Formally, for each $Method \in \{Add, Remove, Contains\}$, we check that $Method(key) \Leftrightarrow SeqMethod(key)$.

Another key property of interest checks if the list nodes refine the specification set S. This invariant is referred to as the *refinement map*. The refinement map states that S matches with the set of values of nodes reachable from the head node, the $ConcSet$. Formally, the refinement map is then stated as: $\forall v. v \in S \Leftrightarrow v \in ConcSet$, where $ConcSet = \{\bigcup_{\forall n \in \widehat{\mathcal{R}}_H} : n.key\}$.

Finally, another key property is the *ordering* invariant. This states that if n_1 is the successor of n_2, then the key of n_1 is greater than the key of n_2.

3 Abstraction

In this section we show how the unbounded threads and list nodes can be abstracted to obtain a finite model.

3.1 Abstracting Unbounded threads

We abstract the unbounded number of threads by using *data type reduction* [12]. This abstraction keeps thread 1 unchanged and creates an environment thread *Other* representing threads $[2..N]$. The abstraction operation involves throwing away all the state variables of threads $[2..N]$ and over-approximating expressions in the guards involving them. For instance $pc[i] = 12$ for $i \in [2..N]$ is thrown away and replaced by *true* or *false* (depending on which replacement leads to an over-abstraction). Next, the action in the transitions of thread *other* may refer to the heap using local pointers. Since the thread *other* is stateless, the local pointers do not have any stored value. Then, the local pointer values are non-deterministically chosen to complete the actions. This abstraction is completely syntactic in nature [18].

Example Consider for example, the transition corresponding to line 12 in Fig 3a, the Murphi rule for which is shown in Fig 3b. In the constructed abstraction, the transition for thread 1 is obtained by substituting i with 1 in the rule . Next, for the *other* thread, the transition is obtained by first over-abstracting the guard with $true$ and second, by replacing $p_0[i]$ and $p_2[i]$ by non-deterministic pointers Np_0 and Np_2. The obtained rule is as follows:
$\forall Np_0, Np_2 \in List : (true) \rightarrow \{\ Np_0.next := Np_2;\ \}$.

3.2 Abstracting Unbounded List

Abstracting the state: The list abstraction consists of two components: first, the shape of the list, and second, the values (key values in particular) of the nodes. These are discussed below:

Shape Abstraction Since the unbounded number of threads have been abstracted to only a single thread, certain nodes in the model may not be reachable from the pointer variables of thread 1 or the global pointers. These nodes are discarded and replaced by a representative node $\widehat{nd}$. [7] The shape abstraction of the remaining heap proceeds by replacing all maximally uninterrupted chains of nodes by *abstract* nodes.

Value Abstraction: In order to prove the refinement map, the correspondence of $ConcSet$ and S must be checked. This requires defining key values for the abstract node also. This is done by replacing D with D^{abs}, where D^{abs} is an interval set induced on D and has a comparison operator. In D^{abs} $(a, b) < (c, d)$, if $b < c$ in D.

Specification set abstraction: Correspondingly, the specification set S is also abstracted to S^{abs} with values from D^{abs}. The sequential methods are also abstracted: as an example, $seqContains$ is abstracted to $seqContains^{abs}$. $seqContains^{abs}(e)$ is specified for e being a singleton set only. It returns *true* if $e \in S^{abs}$ and *false* if $\forall interval \in S^{abs} : e \notin interval$. Otherwise, it has a non-deterministic behavior. The methods of S^{abs} are straightforward and are provided in Appendix A.4.

Example: Fig 5 shows the state obtained after doing the list abstraction followed by thread abstraction of Fig 4. The node corresponding to node 4 is thrown away. Next,

[7] Nodes which become a part of $\widehat{nd}$ can, in practice, never become reachable from any thread pointer again. This is true for the list-based concurrent set data structures we have seen in [8] and is discussed in the Appendix A.2.

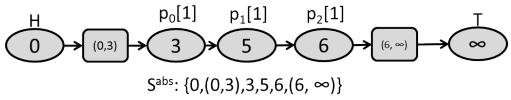


Fig. 5: Abstracting unbounded nodes: maximally uninterrupted nodes replaced by abstract nodes (rectangles).

the node 1 and nodes 7-9 are replaced by abstract nodes. Finally, the key values as well as the values of the abstracted specification set S^{abs} are replaced by values from the interval set induced on $[0, \infty)$.

Abstracting the transitions: A key requirement for abstracting the transitions is that they over-abstract the original transitions and they transition from one valid abstract state to another. The abstracted transitions proceed in 3 phases: (1) a setup phase where the abstract nodes are split if required, (2) a transition phase, where the transition from the *original* program is executed, and (3) a cleanup phase, where after the transition the newly obtained state is lifted to a valid abstract state. We explain these in detail below.

Setup phase: interaction with abstract nodes Since the list abstraction is intuitively designed to have threads accessing only concrete nodes, any access to an abstract node is resolved by non-deterministically splitting the abstract node into a chain of concrete and abstract nodes. The accessing thread then accesses the created concrete node.

Accesses to an abstract node happen in two cases:

(a) *Access to $next$ field of a node with an abstract node as successor.* This, is resolved by, intuitively, selecting the leftmost node in the chain of concrete nodes represented by the abstract node. This is implemented by the GETNEXT function shown in Fig 7a. The GETNEXT function returns the next value by splitting the next abstract node (pointed by $p.next$) non-deterministically in the following two cases: (1) it splits the abstract node into a concrete node followed by an abstract node, and (2) it assumes that the abstract node represents exactly one concrete node and so replaces it by a concrete node. This is accomplished by calling the SPLIT method. The SPLIT method also assigns non-deterministic field values to the newly created nodes by calling the ASSIGNKEYS function. Fig 6 shows how this works for our running example.

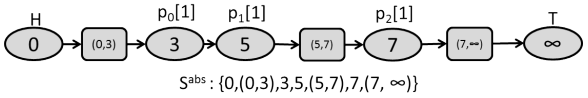


Fig. 6: List traversal by thread 1 by the operation: $p_2[1] := getNext(p_2[1])$. A new concrete node with key 7 and an abstract node with key $(7, \infty)$ are created. The node with key 6 becomes an abstract node.

(b) *Thread other accesses an abstract node:* In this case the abstract node is split by calling the GETNODE function shown in Fig 7b. This function splits the abstract node

```
GETNEXT(p)
  if p.next is concrete
    return p.next;
  else
    n, n_1 := new nodes;
    Switch(non-det value v ∈ {1, 2})
    Case 1:
      SPLIT (p.next, (n, n_1));
    Case 2:
      SPLIT (p.next, (n));
    return n;
  END
```

(a)

```
SPLIT (absNode, n_list)
  pred, succ := predecessor, successor
      of absNode;
    replace pred → absNode
          → succ by
      pred → n_list_0 → ...
          n_list_k → succ;
    S^abs := S^abs − absNode.key;
      ASSIGNKEYS (n_list);
  END
```

(c)

```
GETNODE(p_i[o])
  if p_i[o] is concrete
    return p_i[o];
  else
    n, n_1, n_2, n_3 := new nodes;
    Switch(non-det value v ∈ {1, 2, 3, 4})
    Case 1:
      SPLIT (p_i[o], (n_1, n, n_2));
      return n;
    Case 2:
      SPLIT (p_i[o], (n_1, n));
      return n;
    Case 3:
      SPLIT (p_i[o], (n, n_2));
      return n;
    Case 4:
      SPLIT (p_i[o], (n));
      return n;
  END
```

(b)

```
ASSIGNKEYS (node_list)
  for each node n ∈ node_list
    assign non-det value to n.key;
    S^abs := S^abs + n.key;
  END
```

(d)

Fig. 7: The GETNEXT and GETNODE methods.

non-deterministically in one of four ways: (1) it assumes that the concrete node to be returned on splitting is the leftmost node and so splits into a concrete node followed by an abstract node, (2) it assumes that the concrete node is the rightmost node and so splits into an abstract node followed by the concrete node, (3) it assumes that the concrete node is some central node and so splits into an abstract node followed by the concrete node followed by another abstract node, and (4) it assumes that abstract node represents exactly one concrete node and so replaces it by the concrete node. Non-deterministic values are assigned to the newly created nodes by the SPLIT function by calling ASSIGNKEYS. Fig 8 shows how this is done for our running example.

Transition phase Since all the pointers now refer to concrete nodes, and further, the reads of the next nodes of any pointers also are concrete nodes, the transitions can be executed as though they were executing on the original program with no list abstraction. This is essential in enabling syntactic construction of the list abstraction, as discussed in Section 4.2.

Cleanup phase: canonicalizing to a valid abstract state Once the transition executes as described above, the new state may not be a valid abstract state; i.e., there may be nodes which are not pointed to by any thread pointer which need to be replaced by abstract nodes. Further, there may be nodes which are not reachable from any pointer in the system. The mapping of the new state to a valid abstract state is accomplished by calling the CANONICALIZE method at the end of the transition, as shown in Fig 9.

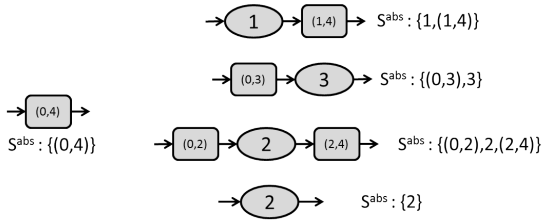


Fig. 8: Splitting of abstract node when a pointer of the *other* thread non-deterministically accesses it. There are four possibilities: the concrete node (1) is the rightmost node (2) is the leftmost node (3) is some node in middle or (4) is the only node in the chain of nodes represented by the abstract node.

Example: As an example, consider the transition corresponding to line 3 in Fig 3a, the corresponding Murphi rule for which is shown in Fig 3b (rule with $pc[i] = 3$). For that transition, on doing the list abstraction to the rule for thread 1 (which is obtained by thread abstraction), the following rule is obtained:

$(pc[1] = 3) \rightarrow \{\ p_1[1] := getNext(p_0[1]);\ pc[1] + +;$
CANONICALIZE (); $\}$

For the *other* thread, the above rule becomes a no-op since no update to the heap is involved.

Next, for the rule corresponding to $pc[i] = 12$, the rule obtained by doing a list abstraction of the rule for thread *other* (obtained after thread abstraction) is:

$\forall Np_0, Np_2 \in List : (true) \rightarrow \{\ getNode(Np_0).next := getNode(Np_2);$
CANONICALIZE (); $\}$.

CANONICALIZE:
  $\mathcal{P}$: Set of all thread local & global pointers
  Throw away nodes $n \notin \widehat{\mathcal{R}}_{\mathcal{P}}$;
  for each $n_i, n_j \in \widehat{\mathcal{R}}_{\mathcal{P}}$
    if (ISMUINT (n_i, n_j))
      JOIN (n_i, n_j);

JOIN (n_i, n_j):
  pn_i := predecessor of n_i;
  sn_j := successor of n_j;
  $pn_i.next := n_{ij}^A$;
  $n_{ij}^A.next := sn_j$;
  for each node n in $n_i \rightarrow ... \rightarrow n_j$
    $S^{abs} = S^{abs} - n.key$;
  ASSIGNKEYS (n_{ij}^A);

ISMUINT (n_i,n_j):
  pn_i := predecessor of n_i;
  sn_j := successor of n_j;
  if $(Ref(pn_i) \& Ref(sn_j))$
    node $n := pn_i$;
    while($n.next! = sn_j$) do
      $n := n.next$;
      if($Ref(n)$) then
        return false;
    return true;
  else
    return false;

Fig. 9: CANONICALIZE Method

Syntactic construction of the abstraction: The thread and list abstractions can be constructed syntactically. The syntactic construction of the thread abstraction is discussed in our previous work [18]. The syntactic construction of the list abstraction can

be done in the following 3 steps: (1) replace accesses to all nodes by pointer Np of the *other* thread with GETNODE (Np). (2) Next, replace all reads to the next field of any node, say of the form $p.next$, with GETNEXT (p). (3) Finally, at the end of each transition, insert a call to CANONICALIZE () function.

While the syntactic construction of the thread abstraction has been implemented in a tool called Abster [19], we have not extended the tool for list-abstraction. This is part of our ongoing work.

4 Refining the abstraction: extended CMP method

The above defined abstraction may be too coarse to prove the property. This may be due to two reasons: (1) the environment thread, *other*, may non-deterministically execute transitions which spuriously violate the property, and (2) the method ASSIGNKEYS called by SPLIT and JOIN may non-deterministically assign key values to the newly created nodes which violate the property. Thus, to prove the property, the above abstraction may have to be refined either in the thread dimension (refining *other*) or in the list dimension (by constraining values assigned by ASSIGNKEYS).

The abstraction and refinement is then done in a loop which is shown in Fig 2. The loop proceeds by iteratively model checking the system model. If the system model at any stage in the refinement loop passes the model checker, the property is proven. If, on the other hand, there is a counterexample for the system model, the user must examine the counterexample. In case the counterexample is valid, a bug has been found. On the other hand, if the counterexample is invalid, the user needs to distinguish between two possible cases. 1) The spurious counterexample is caused due to a spurious transition non-deterministically executed by the *other* thread. 2) The spurious counterexample is due to a new node introduced by the ASSIGNKEYS method. We discuss both these cases in detail below.

4.1 Refinement in thread dimension

Since the environment thread *other* non-deterministically selects nodes and executes transitions, it may exhibit spurious behaviors. In order to constrain the *other* thread, the user adds candidate lemmas which are conjoined with the guards of the rules. Formally, suppose that the candidate lemma L is used. Now consider a rule r of the program P defined as: $\rho \Rightarrow a$, where ρ is the guard and a is an action. Then, refining P with L involves strengthening the guard by L to obtain the strengthened rule $\rho \wedge L \Rightarrow a$, and then re-abstracting the new program with the new rule obtained. Observe that in this refinement approach, no extra state gets added to the abstract model. This is important for efficiency in model checking.

Example: Consider our example for rule with $pc = 12$, for which the abstracted rule for thread *other* is discussed in Section 3.2.

Since that rule is highly unconstrained and may lead to a spurious counterexample, the user may add the lemma that $p_2[i]$ is the successor of $p_1[i]$ which is the successor of $p_0[i]$. This lemma can be expressed as $p_0[i].next = p_1[i]$ & $p_1[i].next = p_2[i]$. Thus, the *strengthened* rule is now:

$(pc[i] = 12$ & $p_0[i].next = p_1[i]$ & $p_1[i].next = p_2[i]) \rightarrow$ { $p_0[i].next := p_2[i]$; $pc[i] + +$; }

Re-abstracting this strengthened rule leads to a more constrained abstract rule for the *other* thread.

$\forall Np_0, Np_1, Np_2 \in Nodes : (true$ & $Np_0.next = Np_1$ & $Np_1.next = Np_2) \rightarrow$ { $getNode(Np_0).next := getNode(Np_2)$;

CANONICALIZE (); }

4.2 Refinement in list dimension

In order to prevent property violations due to non-deterministic values assigned by ASSIGNKEYS method, the user strengthens the model by adding list lemmas (i.e., invariants on list variables). These lemmas are then used to constrain the values assigned to the newly created node.

This list lemma based strengthening is implemented in the GETNODE and GETNEXT functions by modifying the ASSIGNKEYS function to ASSIGNKEYS', as shown in Fig 10. The ASSIGNKEYS' method checks the added list invariants before assigning the key values and exits the while loop when such values are found. Observe that refinement in the list dimension also does not add extra state to the abstract model.

```
ASSIGNKEYS' (node_list)
  while(true)
    for (node n ∈ node_list)
      assign non-det value to n.key and update S;
    if (list lemmas satisfied)
      break;
  END
```

Fig. 10: The ASSIGNKEYS' method

Example: As an example, the keys of the nodes are assumed to be sorted (ordering invariant). Thus, in case the new node is assigned a value such that the ordering invariant is violated, it will lead to a violation of properties like Linearizability as well. The user then adds the ordering lemma during the refinement process.

We show how the strengthening in the list dimension with the ordering lemma affects the following operations: (1) list traversal by thread 1 through call to GETNEXT and (2) list updates by thread *other* by call to GETNODE.

Fig 6 shows how after list traversal ($p_2[1] := p_2[1].next$) from the original state shown in Fig 5, the key value of the newly created nodes respect the ordering lemma.

Next, Fig 8 shows that when the *other* thread accesses an abstract node, the node is split non-deterministically in one of four ways. Observe that after splitting, the values to the newly created nodes are assigned such that the ordering lemma holds.

Proof of Correctness: The correctness of the refinement in the list dimension can be established by the following theorem, the proof for which is provided in Appendix A.3. [8]

[8] For the proof of correctness of the refinement in the thread dimension, we refer the interested reader to [19].

Theorem. *Let P be the original program, let P^{abs} be the abstracted program and let P_g^{abs} be the program obtained by strengthening the list with list invariants $g1, g2, ...$ Then, if the strengthened program P_g^{abs} satisfies $\phi \wedge g$, where g is $g1 \& g2 \& g3 \ldots$ and ϕ is any invariant, the original program P also satisfies $\phi \wedge g$.*

Note that the above theorem implies that if P_g^{abs} satisfies $\phi \wedge g$, the added list lemmas, g, hold for P as well. This shows the soundness of our extension in the list dimension.

5 Experiments

We verified properties of the *Fine-grained* and *Optimistic* algorithms presented in [8] using our method. The *Optimistic* algorithm differs from the *Fine-grained* algorithm in the sense that it reduces contention by traversing the list without locking.

We modeled the two concurrent data structures and their access methods in the Murphi language. The abstraction was implemented by hand. This step is fully automatable but the abstraction tool we used in our earlier work [18] does not yet handle lists (this extension is part of our ongoing work). The strengthening was carried out manually as well.

Verifying Linearizability As discussed in Section 2.2, we verify Linearizability by a refinement based approach.

Added Lemmas: The list lemmas which had to be added to verify Linearizability were: (1) the ordering lemma, which states that the ordering invariant holds, and (2) the refinement lemma [9], which states that the refinement map holds.

Next, in order to refine the thread dimension, the following (classes of) thread lemmas were added. (1) Lemmas specifying relationships between thread pointers while the thread makes updates. For example, for line 12 of the *Remove* method, a lemma stating that the node pointed to by $p_1[i]$ is next to that pointed to by $p_0[i]$ was added. And (2), synchronization lemmas for stating that certain updates be made only when the node pointed to by the pointer is locked by that thread. As an example, in lines 11 and 12 of *Remove* method, the pointers $p_0[i]$, $p_1[i]$ and $p_2[i]$ should have the nodes they point to locked. In all, we had to add 5 thread lemmas for *Fine-grained* algorithm and 6 for the *Optimistic* algorithm.

Runtime: The model checking of the *Fine-grained* algorithm took about 0.97 hours to finish, with 463385 states explored and 344365137 rules fired. The model checking of the *Optimistic* algorithm on the other hand took 24 hours with 6851860 states and 7591017821 rules. [10]

The increased model checking time is due to the large number of fired rules and state space explored. This is due to the large number of interleavings due to fine-grained synchronization and is consistent with similar scalability challenges faced by other model checking efforts in this domain [4, 22, 24, 25].

[9] The refinement lemma is not explicitly added to ASSIGNKEYS'; it is implicitly a part of SPLIT and JOIN. This special treatment is given to it because, strictly speaking, it is a lemma across both, the list and the specification set S.

[10] We did some manual optimizations for the *Optimistic* algorithm to reduce the maximum number of list nodes in the abstraction, in order to reduce the runtime.

Verifying Ordering Since just verifying the ordering invariant does not require checking refinement against S, there is no need to prove the refinement map. This simplifies the proof as the pointers H and T can be dropped. Thus the abstraction size is significantly reduced.

The runtime for checking ordering with above approach was about 79 secs for *Fine-grained* algorithm, with 19620 explored states and 2264535 fired rules. For the *Optimistic* algorithm, the runtime was about 35 min., with 408834 states and 143706285 rules.

While no list lemmas had to be added to verify ordering, 6 lemmas had to be added to constrain the *other* thread for *Fine-grained* and 7 for *Optimistic* algorithms. Note that most of the non-interference lemmas added for proving Linearizability got reused for proving ordering as well.

6 Conclusion and Future Work

We have presented a powerful approach for verifying concurrent list-based data structures and have successfully applied it to verify challenging examples. The key advantage of our approach is that, though it involves some manual guidance, it is largely automatic and it scales to both unbounded number of threads and list nodes.

There are two natural directions in which our work can be extended. Firstly, we can minimize the number of user supplied lemmas by automatically discovering useful lemmas similar to the approach we take in [18]. Secondly, the model checking time for verifying Linearizability can be reduced by designing more efficient abstractions. These abstractions can be used as a part of our overall abstraction-refinement based extended CMP method approach. We plan to take this up next.

In conclusion, our approach opens up a new way for verifying list-based concurrent data structures, which thus far have been handled mainly by separation logic. The preliminary experimental results presented in this paper clearly establish our approach as an alternative model checking based method to verify such data structures.

References

1. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. pp. 399–413. CAV '08, Springer-Verlag, Berlin, Heidelberg (2008)
2. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. pp. 330–340. PLDI '10, ACM, New York, NY, USA (2010)
3. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: In SAS. LNCS. The MIT Press (2007)
4. Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Computer Aided Verification. pp. 465–479 (2010)
5. Chou, C.T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Proc. FMCAD) (2004)

6. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set. In: In 18th CAV. pp. 475–488. Springer (2006)
7. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. SIGPLAN Not. 42(6), 266–277 (Jun 2007)
8. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (July 1990)
10. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Logic 9 (December 2007)
11. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. pp. 321–337. FM '09, Springer-Verlag, Berlin, Heidelberg (2009)
12. McMillan, K.L.: Verification of infinite state systems by compositional model checking. pp. 219–234. CHARME '99, Springer-Verlag, London, UK, UK (1999)
13. Michael, M.M., Scott, M.L.: Correction of a memory management method for lock-free data structures. Tech. rep., Rochester, NY, USA (1995)
14. Noll, T., Rieger, S.: Verifying dynamic pointer-manipulating threads. In: Proceedings of the 15th international symposium on Formal Methods. pp. 84–99. FM '08, Springer-Verlag, Berlin, Heidelberg (2008)
15. O'Leary, J., Talupur, M., Tuttle, M.: Protocol verification using flows: An industrial experience. In: Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. pp. 172 –179 (nov 2009)
16. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. pp. 82–97. TACAS 2001, Springer-Verlag, London, UK (2001)
17. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infty)-counter abstraction. In: Proceedings of the 14th International Conference on Computer Aided Verification. pp. 107–122. CAV '02, Springer-Verlag, London, UK, UK (2002)
18. Sethi, D., Talupur, M., Schwartz-Narbonne, D., Malik, S.: Parameterized model checking of fine grained concurrency. In: 19th International SPIN Workshop on Model Checking of Software (2012)
19. Talupur, M., Tuttle, M.: Going with the flow: Parameterized verification using message flows. In: Formal Methods in Computer-Aided Design, 2008. FMCAD '08. pp. 1–8 (nov 2008)
20. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. pp. 335–348. VMCAI '09, Springer-Verlag, Berlin, Heidelberg (2009)
21. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. pp. 129–136. PPoPP '06, ACM, New York, NY, USA (2006)
22. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software. pp. 261–278. Springer-Verlag, Berlin, Heidelberg (2009)
23. Yahav, E.: Verifying safety properties of concurrent java programs using 3-valued logic. In: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 27–40. POPL '01, ACM, New York, NY, USA (2001)
24. Zhang, S.J.: Scalable automatic linearizability checking. In: Proceeding of the 33rd international conference on Software engineering. pp. 1185–1187. ICSE '11, ACM, New York, NY, USA (2011)
25. Zhang, S.J., Liu, Y.: Model checking a lazy concurrent list-based set algorithm. In: Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. pp. 43–52. SSIRI '10, IEEE Computer Society, Washington, DC, USA (2010)

A Appendix

A.1 Murphi Encoding of the complete *Remove* method

The Murphi encoding of the complete *Remove* method is shown in Fig 11.

$\forall i \in Threads : (pc[i] = 1) \rightarrow \{$
$p_0[i] := H;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 2) \rightarrow \{$
$p_0[i].lock();$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 3) \rightarrow \{$
$p_1[i] := p_0[i].next;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 4) \rightarrow \{$
$p_1[i].lock();$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 5) \rightarrow \{$
$if(p_1[i].key < Key);$
$pc[i] + +;$
$else\ pc[i] = 10;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 6) \rightarrow \{$
$p_0[i].unlock();$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 7) \rightarrow \{$
$p_0[i] := p_1[i];$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 8) \rightarrow \{$
$p_1[i] := p_0[i].next;$
$pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 9) \rightarrow \{$
$p_1[i].lock();$
$pc[i] := 5; \}$

$\forall i \in Threads : (pc[i] = 10) \rightarrow \{$
if $(p_1[i].key = Key)$
then $pc[i] + +;$
else $pc[i] = 15; \}$

$\forall i \in Threads : (pc[i] = 11) \rightarrow \{$
$p_2[i] := p_1[i].next; pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 12) \rightarrow \{$
$p_0[i].next := p_2[i]; pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 13) \rightarrow \{$
$result[i] := true;$
$pc[i] := pc[i] + 3; \}$

$\forall i \in Threads : (pc[i] = 14) \rightarrow \{$
[else] $pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 15) \rightarrow \{$
$result[i] := false; pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 16) \rightarrow \{$
$p_0[i].unlock(); pc[i] + +; \}$

$\forall i \in Threads : (pc[i] = 17) \rightarrow \{$
$p_1[i].unlock(); \}$

Fig. 11: Murphi model for method *Remove* for thread i: each rule corresponds to a line in the *Remove* method shown in Fig 3a.

A.2 Nodes in $\widehat{nd}$

Nodes in $\widehat{nd}$ are those nodes which have been allocated but have become unreachable from any node in the system. In this section we explain why nodes in $\widehat{nd}$ can never

become reachable in the system again and thus affect the abstraction state space. Not that this condition gets automatically verified in the model checking because of the relationship lemma, discussed in Section 5.

We informally explain the reasoning for this as applied to the *Remove* method of the *Fine-grained* algorithm. For $\widehat{nd}$ to become reachable again, there must be a node n, such that n is reachable in the system and $n.next$ is assigned to $\widehat{nd}$. Since the relationship lemma holds, there must exist a node n_1, such that $n.next$ is n_1 and $n_1.next$ is $\widehat{nd}$. But, if this holds, $\widehat{nd}$ is reachable from n and thus reachable in the system. Since this is a contradiction, such a node n_1 does not exist.

A.3 Proof of correctness

Theorem. *Let P be the original program, let P^{abs} be the abstracted program and let P_g^{abs} be the program obtained by strengthening the list with list invariants $g1, g2, ...$ Then, if the strengthened program P_g^{abs} satisfies $\phi \wedge g$, where g is $g1 \wedge g2 \wedge g3 \ldots$ and ϕ is any invariant, the original program P also satisfies $\phi \wedge g$.*

Proof. We show that any counter-example in the original program is also a counter-example in P_g^{abs}.

Suppose the original program does not satisfy $\phi \wedge g$ and let $s_0 \rightarrow^{t_0} s_1 \rightarrow^{t_1} \ldots s_{n-1} \rightarrow^{t_{n-1}} s_n$ be the counterexample, where t_i are transitions and s_i are states. Further, the counterexample is assumed to be of minimal length, i.e., s_n is the first state in which $\phi \wedge g$ is violated.

We prove this in two parts: first we show that the correct part of the counterexample cane be simulated (part 1) and next, we show that the bug is also simulatable (part 2).

Part 1: We first show that $s_0 \rightarrow^{t_0} s_1 \rightarrow^{t_1} \ldots s_{n-1} \rightarrow^{t_{n-2}} s_{n-1}$ can be simulated in the program. We prove this by induction on the counterexample trace. Suppose $s_0 \rightarrow^{t_0} s_1 \rightarrow^{t_1} \ldots s_{i-1} \rightarrow^{t_{i-1}} s_i$ can be simulated in P_g^{abs}. We show that the transition t_i can also be simulated. Suppose t_i involves reading or writing to nodes n_1, n_2..., n_k in the original program. Since $i < n$ and the counterexample is minimal, the list satisfies g. This includes the nodes n_1, n_2..., n_k.

For the corresponding execution in P_g^{abs}, some of the nodes from n_1, n_2..., n_k might be created from splitting abstract nodes. In order to show that the transition t_i is possible in P_g^{abs}, we need to show that if any nodes are created from splitting the abstract nodes, they can still take the field values which are taken by these nodes in the original program. In other words, we need to show that constraining the nodes obtained by splitting abstract nodes to satisfy g by ASSIGNKEYS' does not constrain the nodes to take values taken by n_1, n_2, ... in P. But, since n_1, n_2, ... satisfy g in the original program, constraining by g in P_g^{abs} will not prevent the newly created nodes from taking these values. Thus, the transition will be simulated.

Part 2: Next we show that a violation of t_{n-1} will also be simulated, assuming the rest of the counterexample has been simulated thus far. Without loss of generality, by symmetry, we assume that t_{n-1} is a transition of thread 1. Now, if this happens, we have the following 2 cases:

Case 1: The property ϕ is violated. In this case, like in part 1, assumption of g by ASSIGNKEYS' does not rule out any transitions. Thus, the error-prone transition violating ϕ will be simulated, leading to a violation of ϕ in P_g^{abs} as well.

Case 2: The list-lemma g is violated. In this case, we need to show that the violation occurs and occurs at the transition phase. The violation occurs, due to the same reasoning as for case 1. We show that it occurs in the transition phase.

In case g is violated, the transition t_{n-1} can not be a heap traversal operation in the program P, since it lead to a violation of a list invariant. This is because list invariant depends only on heap values (and potentially on S for refinement map); but none of these change in list traversal in P.

Further, for heap update transitions for thread 1, in P_g^{abs}, the following two conditions hold: (1) all actions by thread 1 occur on concrete nodes, and (2) the nodes which are referred do not change. Consequently, the setup and cleanup phases of this transition do not make any updates to the heap. Thus the violation of the property will occur exclusively at the transition phase. Then, this violation will be simulated in P_g^{abs} as well.

Thus we have shown that any counter-example in the original program P has a corresponding counter-example in P_g^{abs}. Then, our abstraction-refinement method is sound. □

A.4 Methods of S^{abs}

$seqContains^{abs}(e)$: {
if (e not singleton) unspecified;
else if ($\forall interval \in S^{abs} : e \notin interval$) then return false;
else if ($e \in S^{abs}$) then return true;
else return non-det;}

$seqAdd^{abs}(e)$: {
if (e not singleton) unspecified;
else if ($\forall interval \in S^{abs} : e \notin interval$) then S^{abs}:= S^{abs} + e; return true;
else if ($e \in S^{abs}$) then return false;
else return non-det;}

$seqRemove^{abs}(e)$: {
if (e not singleton) unspecified;
else if ($\forall interval \in S^{abs} : e \notin interval$) then return false;
else if ($e \in S^{abs}$) then S^{abs}:= S^{abs}- {e}; return true;
else return non-det;}

Fig. 12: Methods of the sequential set S^{abs}.