Experience with Model Checking Linearizability

Martin Vechev, Eran Yahav, and Greta Yorsh

IBM T.J. Watson Research Center

Non-blocking concurrent algorithms offer significant performance advantages, but are very difficult to construct and verify. In this paper, we describe our experience in using SPIN to check linearizability of non-blocking concurrent data-structure algorithms that manipulate dynamically allocated memory. In particular, this is the first work that describes a method for checking linearizability with non-fixed linearization points.

1 Introduction

Concurrent data-structure algorithms are becoming increasingly popular as they provide an unequaled mechanism for achieving high performance on multi-core hardware. Typically, to achieve high performance, these algorithms use fine-grained synchronization techniques. This leads to complex interaction between processes that concurrently execute operations on the data structure. Such interaction presents serious challenges both for the construction of an algorithm and for its verification.

Linearizability [11] is a widely accepted correctness criterion for implementations of concurrent data-structures. It guarantees that a concurrent data structure appears to the programmer as a sequential data structure. Intuitively, linearizability provides the illusion that any operation performed on a concurrent data structure takes effect instantaneously at some point between its invocation and its response. Such points are commonly referred to as linearization points.

Automatic verification and checking of linearizability (e.g., [7, 8, 1, 23, 21, 2]) and of related correctness conditions (e.g., [5, 4]) is an active area of research. Most of these methods rely on the user to specify linearization points, which typically requires an insight on how the algorithm operates.

Our study of checking linearizability is motivated by our work on systematic construction of concurrent algorithms, and in particular our work on the PARAGLIDER tool. The goal of the PARAGLIDER tool, described in [22], is to assist the programmer in systematic derivation of linearizable fine-grained concurrent data-structure algorithms. PARAGLIDER explores a (huge) space of algorithms derived from a schema that the programmer provides, and checks each of these algorithms for linearizability.

Since PARAGLIDER automatically explores a space of thousands of algorithms, the user cannot specify the linearization points for each of the explored algorithms. Further, some of the explored algorithms might not have fixed linearization points (see Section 4). This motivated us to study approaches for checking the algorithms also in the cases when linearization points are not specified, and when linearization points are not fixed. We also consider checking of the algorithms using alternative correctness criteria such as sequential consistency.

While [22] has focused on the derivation process, and on the algorithms, this paper focuses on our experience with checking linearizability of the algorithms, and the lessons we have learned from this experience.

1.1 Highly-Concurrent Data-Structure Algorithms

Using PARAGLIDER, we checked a variety of highly-concurrent data-structure algorithms based on linked lists, ranging (with increasing complexity) from lock-free concurrent stacks [20], through concurrent queues and concurrent work-stealing queues [18], to concurrent sets [22].

In this paper, we will focus on concurrent set algorithms, which are the most complex algorithms that we have considered so far. Intuitively, a set implementation requires searching through the underlying structure (for example, correctly inserting an item into a sorted linked list), while queues and stacks only operate on the endpoints of the underlying structure. For example, in a stack implemented as linked list, push and pop operations involve only the head of the list; in a queue implemented as a linked list, enqueue and dequeue involve only the head and the tail of the list.

We believe that our experience with concurrent sets will be useful to anyone trying to check properties of even more complex concurrent algorithms, such as concurrent trees or concurrent hash tables [16] which actually use concurrent sets in their implementation.

1.2 Linearizability and Other Correctness Criteria

The linearizability of a concurrent object (data-structure) is checked with respect to a specification of the desired behavior of the object in a sequential setting. This sequential specification defines a set of permitted sequential executions. Informally, a concurrent object is linearizable if each concurrent execution of operations on the object is equivalent to some permitted sequential execution, in which the real-time order between non-overlapping operations is preserved. The equivalence is based on comparing the arguments of operation invocations, and the results of operations (responses).

Other correctness criteria in the literature, such as *sequential consistency* [15] also require that a concurrent execution be equivalent to some sequential execution. However, these criteria differ on the requirements on ordering of operations. Sequential consistency requires that operations in the sequential execution appear in an order that is consistent with the order seen at individual threads. Compared to these correctness criteria, linearizability is more intuitive, as it preserves the real-time ordering of non-overlapping operations.

In this paper, we focus on checking linearizability, as it is the appropriate condition for the domain of concurrent objects [11]. Our tool can also check operation-level serializability, sequential consistency and commit-atomicity [8]. In addition, we also checked data-structure invariants (e.g., list is acyclic and sorted) and other safety properties (e.g., absence of null dereferences and memory leaks).

Checking linearizability is challenging because it requires correlating every concurrent execution with a corresponding permitted sequential execution (linearization). Note that even though there could be many possible linearizations of a concurrent execution, finding a *single* linearization is enough to declare the concurrent exection correct.

There are two alternative ways to check linearizability: (i) *automatic linearization*—explore all permutations of a concurrent execution to find a permitted linearization; (ii) *linearization points*—the linearization point of each operation is a program statement at which the operation appears to take place. When the linearization points of a concurrent object are known, they induce an order between overlapping operations of a concurrent execution. This obviates the need to enumerate all possible permutation for finding a linearization.

For simpler algorithms, the linearization point for an operation is usually a statement in the code of the operation. For more complex fine-grained algorithms, such as the running example used in this paper, a linearization point may reside in method(s) other than the executing operation and may depend on the actual concurrent execution. We classify linearization points as either *fixed* or *non-fixed*, respectively. This work is the first to describe in detail the challenges and choices that arise when checking linearizability of algorithms with non-fixed linearization points. We use program instrumentation, as explained in Section 4.

1.3 Overview of PARAGLIDER

Fig. 1 shows a high-level structure of PARAGLIDER. Given a sequential specification of the algorithm and a schema, the generator explores all concurrent algorithms represented by the schema. For each algorithm, it invokes the SPIN model checker to check linearizability. The generator performs domain-specific exploration that leverages the relationship between algorithms in the space defined by the schema to reduce the number of algorithms that have to be checked by the model checker.

The Promela model, described in detail in Section 3, consists of the algorithm and a client that non-deterministically invokes the operations of the algorithm. The model records the entire history of the concurrent execution as part of each state. SPIN explores the state space of the algorithm and uses the linearization checker, described in Section 4 to checks if the history is linearizable. Essentially, it enumerates all possible linearizations of the history and checks each one against the sequential specification. This method is entirely automatic, requires no user annotations, and is the key for the success of the systematic exploration process. The main shortcoming of this method is that it records the entire history as part of the state, which means no two states are equivalent. Therefore, we limit the length of the history by placing a bound on the number of operations the client can invoke.

PARAGLIDER supports both automatic checking, described above, and checking with linearization points. The latter requires algorithm-specific annotations to be provided by the user, but allows the model checker to explore a larger state space of the algorithm than the first approach.

The generator produces a small set of candidate algorithms, which pass the automatic linearizability checker. This process is shown in the top half of Fig. 1. The user can perform a more thorough checking of individual candidate algorithms by providing PARAGLIDER with linearization points for each of them. The linearization is built

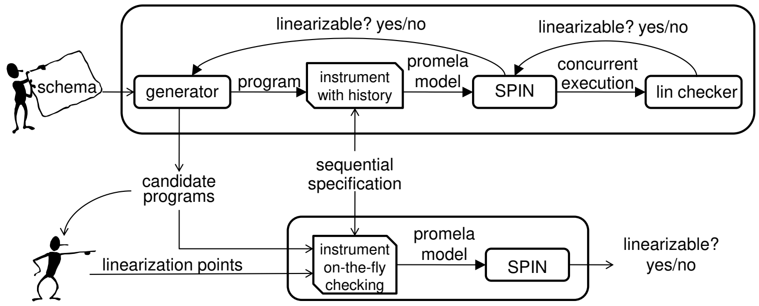

Fig. 1. Overview of PARAGLIDER tool.

and checked on-the-fly in SPIN using linearization points. Thus, we no longer need to record the history as part of the state. This process is shown in the bottom half of Fig. 1.

In both methods, the client is a program that invokes non-deterministically selected operations on the concurrent object. However, with the linearization point method, we can check algorithms with each thread executing this client, *without* any bound on the maximum number of operations (the automatic method requires such a bound).

1.4 Main Contributions

Here we summarize our experience and insights. We elaborate on them in the rest of the paper.

Garbage collection garbage collection (GC) support in a verification tool is crucial for verifying an increasing number of important concurrent algorithms. Because SPIN does not provide GC support, we implemented GC as part of the input Promela model. We discuss the challenges and choices that we made in this process.

Non-fixed linearization points For many advanced concurrent algorithms the linearization point of an operation is *not* in the code of that operation. Checking such algorithms introduces another set of challenges not present in simpler algorithms such as queues or stacks, typically considered in the literature. We discuss the underlying issues as well as our solution to checking algorithms with non-fixed linearization points.

Choosing Bounds We discuss how we chose the bounds on the size of the heap in states explored by SPIN, and how this choice is related to the optimistic algorithms we are checking. We discuss different methods for checking linearizability and how each method inherently affects the size of the state space the model checker can explore.

Data structure invariants vs. Linearizability We discuss our experience in finding algorithms that are linearizable, but do not satisfy structural invariants. This motivates further work on simplifying formal proofs of linearizable algorithms.

Sequential consistency vs. Linearizability We discuss our experience in finding concurrent data structure algorithms which are sequentially consistent, but not linearizable.

2 Running Example

```
boolean add(int key) {
 Entry *pred,*curr,*entry;
restart:
 LOCATE(pred, curr, key)
 k = (curr->key==key)
 if (k) return false
 entry = new Entry(key)
 entry->next = curr
 atomic {
  mp = !pred->marked
  val = (pred->next==curr)
     &&mp
  if (!val) goto restart
  pred->next = entry
 }
 return true
}
```

```
boolean remove(int key) {
 Entry *pred,*curr,*r
restart:
 LOCATE(pred, curr, key)
 k= (curr->key==key)
 if (k) return false
 curr->marked = true
 r = curr->next
 atomic {
  mp = !pred->marked
  val = (pred->next==curr)
     &&mp
  if (!val) goto restart
  pred->next = r
 }
 return true
}
```

```
boolean contains(int key) {
 Entry *pred,*curr;
 LOCATE(pred, curr, key)
 k=(curr->key==key)
 if (!k) return false
 if (k) return true
}

LOCATE(pred, curr, key) {
 pred=Head
 curr=Head->next
 while(curr->key < key) {
  pred=curr
  curr=curr->next
 }
}
```

Fig. 2. A set algorithm using a marked bit to mark deleted nodes. A variation of [9] that uses a weaker validation condition.

To illustrate the challenges which arise when checking linearizability of highly-concurrent algorithms, we use the concurrent set algorithm shown in Fig. 2.

This algorithm is based on a singly linked list with sentinel nodes *Head* and *Tail*. Each node in the list contains three fields: an integer variable *key*, a pointer variable *next* and a boolean variable *marked*. The list is intended to be maintained in a sorted manner using the *key* field. The *Head* node always contains the minimal possible key, and the *Tail* node always contains the maximal possible key. The keys in these two sentinel nodes are never modified, but are only read and used for comparison. Initially, the set is empty, that is, in the linked list, the *next* field of *Head* points to *Tail* and the next field of *Tail* points to *null*. The *marked* fields of both sentinel nodes are initialized to *false*. This algorithm consists of three methods: `add`, `remove` and `contains`.

To keep the list sorted, the `add` method first optimistically searches over the list until it finds the position where the key should be inserted. This search traversal (shown in the `LOCATE` macro) is performed optimistically without any locking. If a key is already in the set, then the method returns *false*. Otherwise, the thread tries to insert the key. However, in between the optimistic traversal and the insertion, the shared invariants may be violated, i.e., the key may have been removed, or the predecessor which should point to the new key may have been removed. In either of these two cases, the algorithm does not perform the insertion and restarts its operation to traverse again from the beginning of the list. Otherwise, the key is inserted and the method returns *true*. The operation of the `remove` method is similar. It iterates over the list, and if it does not find the key it is looking for, it returns *false*. Otherwise, it checks whether the shared invariants are violated and if they are, it restarts. If they are not violated, it physically removes the node and sets its *marked* field to *true*. The *marked* field and setting it to *true* are important because they consistute a communication mechanism to tell other threads that this node has been removed in case they end up with it after the optimistic traversal.

The last method is `contains`. It simply iterates over the heap without any kind of synchronization, and if it finds the key it is looking for, it returns *true*. Otherwise, it returns *false*.

It is important to note that when `add` or `remove` return *false*, they do not use any kind of synchronization. Similarly, for the `contains` method. That is, these methods complete successfully *without* using any synchronization, even though as they iterate, the list can be modified significantly by `add` and `remove` operations executed by other threads. It is exactly this kind of iteration over the linked list *without* any synchronization that distinguishes the concurrent set algorithms from concurrent stack and queues, and makes verification of concurrent sets significantly more involved.

Memory Management This algorithm requires the presence of a garbage collector (GC). That is, the memory (the nodes of the linked list) is only managed by the garbage collector and not via manual memory management. To understand why this particular algorithms requires a garbage collector, consider the execution of the `remove` method, right after the node is disconnected from the list, see line 31. It would be incorrect to free the removed node immediately at this point, because another thread may have a reference to this node. For example, a `contains` method may be iterating over the list optimistically and just when it is about to read the *next* field of a node, that node is freed. In this situation, `contains` would dereference a freed node — a memory error which might cause a system crash. There are various ways to add manual memory management to concurrent algorithms, such as hazard pointers [17]. However, these techniques complicate the algorithm design even further.

Practically, garbage collection has gained wide proliferation via managed languages such as Java, X10, C#. In addition, as part of their user-level libraries, these languages provide a myriad of concurrent data-structure algorithms relying on GC. Hence, developing techniques to ensure the correctness of highly concurrent algorithms relying on automatic memory management has become increasingly important.

3 Modeling of Algorithms

We construct a Promela model that is sound with respect to the algorithm up to the bound we explore, i.e., for every execution of the algorithm which respects the bound, in any legal environment, there is also an execution in the model. The goal is to construct an accurate Promela model which is as faithful as possible to the algorithm and its environment (e.g., assumption of a garbage collector). In this section, we explain the main issues we faced when modeling the algorithms.

3.1 Modeling the Heap

The first issue that arises is that our algorithms manipulate dynamically allocated heap memory and linked data structures of an unbounded size. However, the Promela language used by SPIN does not support dynamically allocated memory (e.g. creating new objects, pointer dereference). Our desire was to stay with the latest versions of the SPIN tool, as they are likely to be most stable and include the latest optimizations, such as partial order reductions. Therefore, we decided not to use variants of SPIN such as dSPIN [6], which support dynamically allocated memory, but are not actively maintained. Hence, in order to model dynamically allocated memory, we pre-allocate a global array in the Promela model. Each element of the array is a Promela structure that models a node of a linked data-structure. Thus, pointers are modeled as indices into the array.

3.2 Garbage Collection

As mentioned in Section 2, the algorithms we consider, as well as many other highly-concurrent optimistic algorithms (e.g.,[10]), assume garbage collection. Without garbage collection, the algorithms may leak an unbounded amount of memory, while manual memory management is tricky and requires external mechanisms, such as hazard pointers [17]. Unfortunately, SPIN does not provide garbage collection support. [1] Hence, we define a garbage collector as part of the Promela model.

Naturally, our first intuitive choice was to have a simple sequential mark and sweep collector that would run as a separate thread and would collect memory whenever it is invoked. This approach raises the following issues:

- The collector needs to read the pointers from local variables of all the other threads. Unfortunately, at the Promela language level, there is no mechanism for one thread to inspect local variables of another thread. To address it, we could make these variable shared instead of local.
- The collector needs to know the type of these variables, that is, whether these values are pointers or pure integer values (e.g. does the variable denote the key value of the node or is that the pointer value which is also modeled as an integer?). To address it, we could make the type of each shared variable explicitly known to the collector.
- When should the garbage collection run?

Making all of the thread local variables globally visible, so that the collector process can find them, is not an ideal solution as it may perturb partial order optimizations. Further, if the collector does not run immediately when an object becomes unreachable, it may result in exploring a large number of distinct states that are meaningless. That is, two states may differ only in the different unreachable and not yet collected objects. This hypothesis was confirmed by our experiments with the algorithm in Fig. 2, where even on machines with 16GB of memory, the exploration did not terminate (we tried a variety of choices for SPIN optimizations).

To address this issue, we concluded that garbage collection should run on every pointer update, effectively leading us to implement a reference counting algorithm. Each node now contains an additional field, the *RC* field, which is modified on pointer updates. Once the field reaches zero, the object is collected. The collector runs atomically. Once the object is collected, it is important to clear all of the node fields in order to avoid creating distinct states that differ only in those object fields. Our reference counting collector does not handle unreachable cycles, because in the algorithms we consider (based on singly linked lists), the heap remains acyclic. Acyclicity is checked as part of the structural invariants.

[1] In [13], dSpin was extended with garbage collection, but it has not been adopted in SPIN.

Despite the fact that the size of a single state increases, with a reference counting collector, the total number of states became manageable. To address the issue of increasing state size, we experimented with various optimizations tricks (such as bit-packing all of the object fields). However, at the end we decided against such optimizations as it was becoming quite difficult to debug the resulting models and even worse, was obscuring our understanding of them.

To use this reference counting approach, our models are augmented with the relevant operations on every pointer update statement. This requires careful additional work on behalf of the programmer. It would certainly have saved significant time had the SPIN runtime provided support for dynamically allocated memory and garbage collection. Further, our approach can also benefit from enhancing SPIN with heap and thread symmetry reductions, e.g., [12, 19, 3], to enable additional partial order reductions. Symmetry reduction are very useful in our setting as threads execute the same operations.

3.3 Choosing Bounds on Heap Size

Because our algorithms may use unbounded amount of memory, we need to build a finite state space in order to apply SPIN. Our models are parameterized on the maximum number of keys in the set, rather than the maximum number of objects. The maximum number of keys (and threads) determines the number of objects. The reason is that it is difficult to say a priori what is the maximum number of objects that the algorithm will need. Due to the high concurrency of the algorithm, situations can arise where for two keys we may need for example 10 objects. The reason for that is not completely intuitive, as shown by the following example.

Example 1. Consider a thread executing `LOCATE` of some operation from Fig. 2, on a set that consists of two keys, 3 and 5. Suppose that the thread gets preempted when it is holding pointers to two objects with keys 3 and 5, via its thread local variables `pred` and `curr`, respectively. Second thread then successfully executes `remove(3)` followed by `remove(5)`, removing from the list the objects that the first thread is holding pointers to. Of course, these objects cannot be collected yet, because the first thread is still pointing to them. Then, the second thread executes `add(3)` and `add(5)`, and successfully inserts new objects with the same keys, while the removed objects are still reachable from the first thread. Thus, with only two keys and two threads, we created a heap with 4 reachable objects.

Via similar scenarios, one can end up with a surprisingly high number of reachable objects for a very small number of keys. In fact, initially we were surprised and had to debug the model to understand such situations. Moreover, for different algorithmic variations the maximum number of objects can vary. Of course, we would not want to pre-allocate more memory for objects than is required as this would increase the size of memory required for model checking. Hence, we experimentally determined the maximum number of objects required for a given number of keys. That is, we start with a number K of pre-allocated objects and if the algorithm tries to allocate more than K objects, we trigger an error and stop. Then, we increase the value of K and repeat the process.

4 Checking Linearizability

There are two alternative ways to check linearizability: (i) *automatic linearization*—explore all permutations of a concurrent execution to find a valid linearization; (ii) *linearization points*—build linearization on-the-fly during a concurrent execution, using linearization points provided by the user. While the second approach requires user-provided annotations, it can check much deeper state spaces. In this section, we first review the definition of linearizability, and then describe how both of these approaches are realized in our models.

4.1 Background: Linearizability

Linearizability [11] is defined with respect to a sequential specification (pre/post conditions). A concurrent object is linearizable if each execution of its operations is equivalent to a permitted sequential execution in which the order between non-overlapping operations is preserved.

Formally, an operation op is a pair of invocation and a response events. An invocation event is a triple $(tid, op, args)$ where tid is the thread identifier, op is the operation identifier, and $args$ are the arguments. Similarly, a response event is triple (tid, op, val) where tid and op are as defined earlier, and val is the value returned from the operation. For an operation op, we denote its invocation by $inv(op)$ and its response by $res(op)$. A *history* is a sequence of invoke and response events. A *sequential history* is one in which each invocation is immediately followed by a matching response. A *thread subhistory*, $h|tid$ is the subsequence of all events in h that have thread id tid. Two histories h_1, h_2 are *equivalent* when for every tid, $h_1|tid = h_2|tid$. An operation op_1 precedes op_2 in h, and write $op_1 <_h op_2$, if $res(op_1)$ appears before $inv(op_2)$ in h. A history h is linearizable, when there exists an equivalent sequential history s, called a linearization, such that for every two operations op_1, op_2, if $op_1 <_h op_2$ then $op_1 <_s op_2$. That is, s is equivalent to h, and respects the global ordering of non-overlapping operations in h.

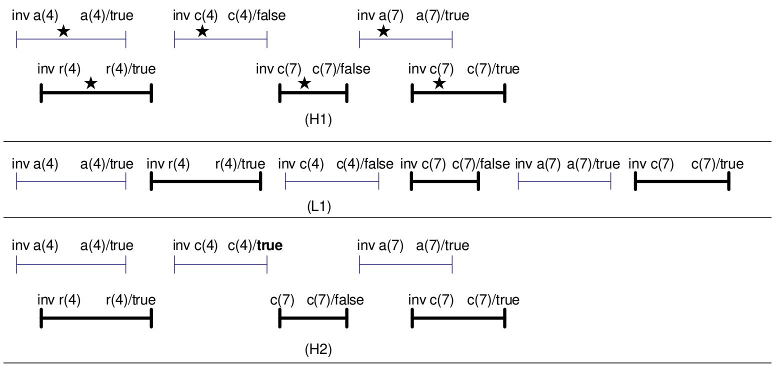

Fig. 3. Concurrent histories and possible sequential histories corresponding to them.

Example 2. Fig. 3 shows two concurrent histories H_1 and H_2, and a sequential history L_1. All histories involve two threads invoking operations on a shared concurrent set. In the figure, we abbreviate names of operations, and use `a`,`r`, and `c`, for `add`, `remove`, and `contains`, respectively. We use `inv op(x)` to denote the invocation of an operation `op` with an argument value `x`, and `op/val` to denote the response `op` with return value `val`.

Consider the history H_1. For now, ignore the star symbols. In this history, `add(4)` is overlapping with `remove(4)`, and `add(7)` overlaps `contains(7)`. The history H_1 is linearizable. We can find an equivalent sequential history that preserves the global order of non-overlapping operations. The history L_1 is a possible *linearization* of H_1 (in general, a concurrent history may have multiple linearizations).

In contrast, the history H_2 is non-linearizable. This is because `remove(4)` returns $true$ (removal succeeded), and `contains(4)` that appears *after* `remove(4)` in H_2 also returns $true$. However, the history H_2 is *sequentially consistent*.

We know from the definition of linearizability that for every operation *op*, there exists a point between its invocation $inv(op)$ and response $res(op)$ in the history h where *op* appears to take effect. This point is typically referred to as the *linearization point* $lp(op)$ of the operation *op*. Given a concurrent history h, the (total) ordering between these points induces a linearization.

Example 3. Consider the history H_1 of Fig. 3, the star symbols in the figure denote the occurrence of a user-specified linearization point in each operation. The relative ordering between these points determines the order between overlapping operations, and therefore determines a unique linearization of H_1, shown as L_1.

4.2 Automatic Linearization

A straightforward way to automatically check whether a concurrent history has a corresponding linearization is to simply try all possible permutations of the concurrent history until we either find a linearization and stop, or fail to find one and report that the concurrent history is not linearizable. We refer to this approach as *Automatic Linearization*. While this approach is conceptually simple, its worst-case time and space complexity is exponential in the length of the concurrent history. Despite its inherent complexity costs, we do use this method for checking concurrent histories of small length (e.g. less than 20). In practice, the space used for incorrect algorithms is typically small because incorrect algorithms often exhibit an incorrect concurrent history that is already almost sequential. We use this approach when we need a fast way to automatically filter through a massive search space, as part of the process of exploring algorithms, but this approach can be used in stand-alone mode as well.

Example 4 (Enumeration). Fig. 4 demonstrates how the checking procedure works for a simple history H1. This history is an example for the kind of histories recorded inside a single state that is explored by the model checker. There are two threads in the history: one thread executes two `add(5)` operations, both of which return $true$ while the second thread executes a `remove(5)` operation that also returns $true$. To check linearizability of H1, our procedure enumerates all potential linearizations of H1, obtained by re-ordering only overlapping operations of H1. For each of these three potential linearizations (shown as L1, L2, and L3), we execute the operations over the (executable) sequential specification of the set, and compare the return value of each operation to its corresponding return value in the concurrent history. In this example, L1 is not a valid linearization of H1 as its `remove(5)` returns $false$. Similarly, L3 is not a valid linearization of H1 as its second `add(5)` returns $false$. The sequential history L2 is a valid linearization of H1 as its invocation and response values match the ones in the concurrent history.

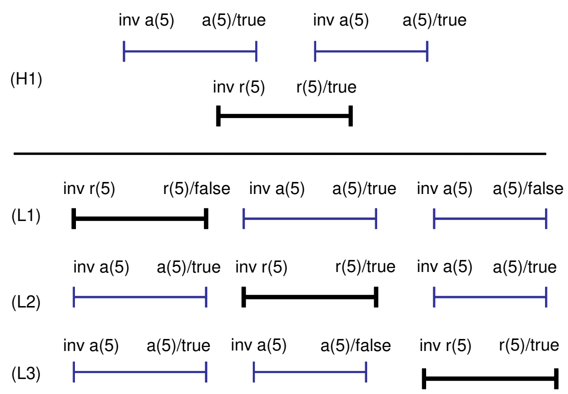

Fig. 4. Enumeration of linearizations. (H1) is a concurrent history, and (L1),(L2), and (L3) are its three potential linearizations.

Instrumentation We instrument the algorithm to keep the executed history as part of the state. That is, each time an operation such as `add(k)` occurs, it records its start event, its argument and its return event (and the value of that return event if any). After the model exploration process completes, we invoke an external checker procedure (the lin. checker component of Fig. 1). This checker takes as input the recorded history and verifies whether there is a sequential witness corresponding to that history.

While automatic and hence useful to filter many algorithms, the main problem with the automatic approach is that every time we append an element into the history, we introduce a new state. Therefore, it is impossible to model check the algorithms using the most general client, which invokes an unbounded number of operations. To make the state space finite, we place a bound on the maximal number of operations that a client executes. This limits the length of the recorded histories.

4.3 Checking Linearizability using Linearization Points

An alternative approach provides time and space complexity which is linear in the length of the concurrent history, but the approach requires the linearization points as input. Usually, the user provides linearization points by explicitly defining them as statements in the code of the algorithm. Typically, these linearization points are at program locations that update a globally visible shared variable. For example, in Fig. 2, the linearization point of a successful `remove` operation returning *true* is the instruction `pred->next = r`.

Once the linearization point of an operation is known, we can instrument the model to check linearization on-the-fly, as follows. We instrument the state to contain a sequential version of the concurrent data-structure, which is maintained independently of the concurrent version in the same state. When we encounter a linearization point during model checking of the concurrent execution, we execute the corresponding sequential operation on the sequential version of the data-structure, and record its result. When the concurrent operation completes, we compare its result to the (recorded) result of the sequential operation.

In the algorithms we checked, the linearization point is immediately before the return of the concurrent operation, and the return value of the concurrent operation is known. Therefore, we can immediately compare the result of the concurrent operation to that of the sequential, without recording the latter.

Non-fixed Linearization Points For many concurrent algorithms, the linearization point of an operation is not fixed, i.e., it may depend on other threads executing concurrently.

Example 5 (Non-fixed Linearization Points). Consider the example concurrent history H1 shown in Fig. 5 in which one thread is performing `remove` with key 5 and another thread is performing a sequence of adding and removing the same key 5. The star symbol denotes a successful linearization point of adding or removing 5 from the set. The linearization point of the `remove` operation which returns false can be placed anywhere in the greyed areas, i.e., anywhere before the linearization point of a successful addition or after the linearization point of a successful removal of key 5 by another thread. Fig. 5 shows the three different valid linearizations of H1: L1, L2, and L3.

In fact, it is impossible to describe the linearization point as a single, fixed, program location for the cases when `add` or `remove` return *false*, and for both return values of `contains`. Intuitively, to describe the linearization point in such cases, we need to take into account the relevant operations performed by other threads. Further, the choice of linearization points is not unique. For example, consider the case of `remove` returning *false*. We can informally describe the linearization point in either one of the following ways:

(1) The linearization point for an executing `remove` operation which returns *false* for key k is the earliest of two events: either before a successful addition of key k by another concurrent thread or the statement at line 22 of `remove`, which compares key k to *curr*'s key in the current execution of `remove`. (Note how the linearization point of one operation can now be found in the execution of another operation.)

(2) The linearization point of `remove` operation that returns *false* for key k is either before a successful addition of key k by another concurrent thread or after a successful removal of key k by another concurrent thread, or the statement at line 22 of

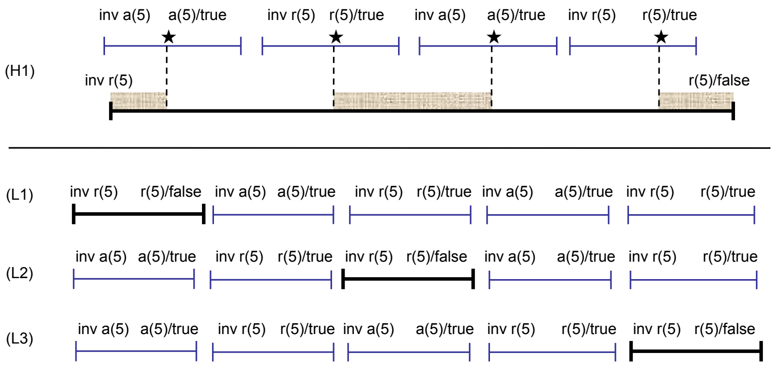

Fig. 5. Non-fixed linearization point of remove: (H1) is a concurrent history, and (L1),(L2), and (L3) are its three valid linearizations.

`remove`, which compares key *k* to *curr*'s key in the current execution of `remove`. (That is, if another thread removes the key *k* that this method is trying to remove, we can linearize the current method at the point right after the key is removed successfully by the other thread.)

The second definition is a more refined than the first, including the possibility of having the linearization point of a failed remove be the linearization point of a successful removal of the same key by another thread. We can keep refining this definition by adding longer sequences of add's and remove's of the same key by other threads.

To summarize, for the cases of `add` and `remove` returning *false*, and for `contains`, not only can the linearization point of an operation be in different thread than the one executing the operation, but there may be multiple choices for choosing these points. Furthermore, small changes in the algorithm can cause it to have dramatically different linearization points. In general, the more fine-grained the algorithm is, the more complex its linearization points are. From a verification standpoint, it is challenging to identify that a non-fixed linearization point is encountered during model checking, and to invoke the corresponding sequential operation.

4.4 Instrumenting for Non-fixed Linearization Points

To deal with non-fixed linearization points, we instrument the program with enough information to know whether another thread is manipulating the list with the same key as the current thread. For example, in the case of the `remove` operation returning *false*, we can now check before the return of `remove` , whether there was another thread executing an overlapping concurrent `add` operation with the same key. If this is true, then we know we can linearize the current `remove` operation before that successful `add` execution. In effect, in the instrumented version of the original program, all the linearization points are once again in the same thread. The instrumentation allows us to observe enough information about actions of other threads and deal with linearization points only in the same thread.

```
/** most general client */
inline execute(){
  do :: {
  select_op_and_key();
  exec_operation(op, key);
  }      :: break; od;
}
inline select_op_and_key(){
 atomic
 {
  if :: op = ADD;
     :: op = REMOVE;
         :: op = CONTAINS; fi;
  }
  get_random_key();
}
/** algorithm **/
inline exec_operation(op, key) {
 start_op();
 do:: (true) -> {
   assert(pred == 0);
   pred = HEAD; // optimized
   get_next_stmt(curr, HEAD);
   assert(curr != 0);
   val_a = get_key(curr);
   retval = RET_RESTART;
    do :: (val_a < key) -> {
       assign(pred, curr);
       get_next_stmt(curr, curr);
       val_a = get_key(curr);
   }
   :: (val_a >= key) ->
       {val_a = 0; break; }
   od;
   if :: (op == ADD) -> {add(key); }
      :: else fi;
   if :: (op == REMOVE) -> {remove(key); }
      :: else fi;
   if :: (op == CONTAINS) -> {contains(key); }
      :: else fi;

ENDOP:
 atomic {
   ... // some cleanup
   if :: (retval != RET_RESTART) ->      {
/*** Linearization Point Checking. */
 seq_inlist(key);
 if :: (op == REMOVE && retval == RET_FALSE) ->
 {assert(i_was_notified(key) || val_f == 1);}
 :: else fi;
 if :: (op == ADD     && retval == RET_FALSE) ->
 {assert(i_was_notified(key) || val_f == 2);}
 :: else fi;
 if :: (op == CONTAINS && retval == RET_FALSE) ->
 {assert(i_was_notified(key) || val_f == 1);}
 :: else fi;
 if :: (op == CONTAINS && retval == RET_TRUE) ->
 {assert(i_was_notified(key) || val_f == 2);}
 :: else fi;
 val_f = 0;
 break;
  }
  :: else val_f = 0 fi;
  };
 } od;
 end_op();
}
```

```
/* instrumentation for non-fixed
   linearization points */
typedef id_t {
 bit val[2];     // 2 processes
};
id_t notify [NODES];
inline  notify_others(k) {
 assert(k < NODES);
 if :: (_pid == 0) -> notify[k].val[1] = 1;
    :: else -> {assert(_pid == 1);
         notify[k].val[0] = 1;}
 fi;
}
#define i_was_notified(k) notify[k].val[_pid]
inline  zero_my_key(k)
{notify[k].val[_pid] = 0;}

inline start_op() {
 atomic {
  init_locals();
  zero_my_key(key);
 };
};
inline end_op() {
 atomic {
  assert(retval == RET_FALSE
         || retval == RET_TRUE);
  init_locals();
  zero_my_key(key);
  op = 0;
  key = 0;
 };
};
inline add(key) {
  ... // make global change
  notify_others(key);
  seq_add(entry, key);
  ...
}

/* executable sequential specification */
inline seq_add(entry, key) {
 check_key(key);
 if :: (seqset[key] == 0) ->
 seqset[key] = 1 :: else ->
 assert(false); fi;
}
inline seq_remove(key){
 check_key(key);
 if :: (seqset[key] == 1) ->
 seqset[key] = 0 :: else ->
 assert(false); fi;
}
inline seq_contains(key, found){
 check_key(key);
 if :: (found == 1) ->
 assert(seqset[key] == 1); :: else ->
 assert(seqset[key] == 0); fi;
}
inline seq_inlist(key){
 if :: (seqset[key] == 1) ->
 val_f = 2; :: else->
 val_f = 1; fi;
}
```

Fig. 6. A fragment of Promela model for checking linearizability using user-specified linearization points, including instrumentation of non-fixed points.

We find it instructive to think about this instrumentation as an abstraction of the concurrent history. This abstraction records for each operation whether there is an overlapping concurrent operation that uses the same key. That is, we do not keep the whole history as part of the state, like the automatic linearization approach does, but only keep an abstraction of that history.

Technically, we instrument every state to record for every thread the operation the thread is executing and with what argument that operation was invoked. Fig. 6 shows a fragment of the Promela model we use for checking linearizability with user-specified linearization points, including instrumentation of non-fixed points. The array `notify` contains for each key and each thread whether another thread is currently executing an operation on that key. A thread `_pid` executing an operation `op` with key `key` first invokes `zero_my_key(key)`, which sets the corresponding entry of the array `notify` to 0. Whenever the concurrent execution encounters a linearization point of a successful `add` or `remove` on an operation, the current thread atomically executes the sequential version and compares the results. Additionally, the thread invokes `notify_others(key)` to set `notify` array of other threads with the same key. On return of an operation returning false, the current thread checks whether it was notified and compares the results.

In general, one should be careful when instrumenting the program in order to use this approach, so to make sure that the additional instrumentation does not affect the behavior of the core model.

Unlike the automatic linearization approach, the linearization points approach does not keep the whole history as part of the state, but only keeps an abstraction of the history. This abstraction is bounded by the number of threads and keys (see Section 3.3). Therefore, unlike the automatic linearization, the linearization points approach does not require an a priori bound on the maximum number of operations executed by the client. In this approach, we model check the algorithm with the most-general client, which executes an unbounded number of operations.

Algorithms such as concurrent stack or queues usually have fixed linearization points. This work advances the state of the art towards dealing with more complex algorithms. In particular, highly-concurrent algorithms that have read-only operations (e.g. `contains` in the concurrent set) are likely to have non-fixed linearization points for these operations, and can be checked using the instrumentation described here. We are not aware of any other tool or published work that is able to automatically check the linearizability of concurrent algorithms with non-fixed linearization points.

5 Experiments

Based on the mechanisms described earlier, we have constructed a large number of Promela models for a variety of concurrent algorithms including: concurrent stacks, concurrent queues, concurrent work-stealing queues, and concurrent sets. For each model, we have checked various properties including linearizability, sequential consistency, structural invariants, and memory safety. We have not discussed sequential consistency in detail in this paper, but our automatic tool that checks linearizability also has the option to check sequential consistency. Sequential consistency is a weaker specification than linearizability, i.e., every linearizable algorithm is sequentially consistent. For structural invariants, we checked that the list remains acyclic (for each run of the client), while for memory safety, we checked that the algorithm does not leak memory.

All our experiments were done on an 8-core 2.4 GHz AMD Opteron with 16GB of memory. For the concurrent set algorithm show in Fig. 2 and its variants, we checked linearizability using the linearization point method (that explores larger state spaces than the automatic one) for two threads and two keys (recall that this may require up to 10 objects). This took around 3GB of memory. We did try to increase the number of threads to three, but the state space was larger than 16GB available on the machine. However, during the time it took SPIN to reach 16GB, it did not detect an error.

There are several variants of the set algorithm shown in Fig. 2. However, for none of them the model checking process with three threads completed. We also tried four and more threads just to see if some error will be triggered in the 16GB available, but no error was triggered.

Next, we observe that all the threads are executing the same operations. Hence, symmetry reduction would have been a very valuable optimization here. Unfortunately, SPIN does not provide symmetry reduction. We did use the Symmetric Spin [3] tool, but for three threads, it still did not complete in 16GB (although the rate of increase for used memory that SPIN outputs did decrease *significantly*). We suspect that if we provide a machine with more memory it may complete and further, for four or more threads, we would not see significant (or any) increases in memory. Our limited experience suggest that including symmetry reduction in the SPIN model checker would certainly be valuable in our context. (SymmSpin is not actively maintained and also requires manual effort which can be error prone).

Next, we experimented with several questions which are interesting both from a verification as well as algorithmic perspectives.

5.1 Linearizability vs. Invariants

In all of our experiments, we checked both structural invariants in addition to linearizability. These checks produced the same results, that is, all algorithms rejected due to linearizability violations also violated memory safety or some structural invariant. This suggests that concurrent algorithms are designed with local structural invariants in mind, rather than global temporal safety properties such as linearizability. Intuitively, this makes sense, as it is difficult to focus on the interleavings of more than few objects at a time. This suggests a new approach for formal verification of concurrent algorithms: instead of proving linearizability directly as recent work has done, it may be beneficial to prove simpler properties such as structural invariants, and then prove that they imply linearizability.

5.2 Linearizability vs. Sequential Consistency

In the context of PARAGLIDER, which generates a massive number of algorithmic variations, we considered sequential consistency as as a simpler filtering criteria, alternative to linearizability. Theoretically, sequential consistency permits more algorithms than linearizability, because sequential consistency does not require preservation of real-time order. Thus, we tried to find algorithms which are sequentially consistent, but not linearizable.

Interestingly, in our experiments, we did not find any such algorithms. We speculate that there is a deeper reason for this result that is worth exploring further. For example, the recent work of [14] formalized the intuitive reasons for preferring linearizability correctness criteria over weaker criteria such as sequential consistency, from the viewpoint of the client, i.e., the program which program uses the concurrent data structure. It would also be interesting to explore this direction from the viewpoint of the algorithm, and not only from the viewpoint of the client.

5.3 Inferring Linearization Points

Linearization points can be often tricky to discover and describe and the latter may require additional instrumentation. Even in the case where the linearization points are fixed, it may not be immediately obvious what these points are. For example, in the algorithm from Fig. 2, the linearization point for a successful `remove` operation is *not* the marking of the node, but the physical removal. This is different than all of the existing concurrent set algorithms whose linearization point is the marking of the current node, e.g. [10, 16]. Further, slight changes to the algorithm may change the linearization points.

One interesting challenge is whether these points can be automatically inferred. Given the program and the specification, can we build an analysis that automatically infers what the linearization points are ? Further, given these points, can we automatically instrument the program in order to perform the necessary model checking ?

Automatic inference would be especially helpful in the case of non-fixed points. Knowing these points is crucial to understanding how the algorithms works. Further, from the point of view of verification, manual instrumentation is potentially error-prone. Automatic inference and instrumentation would thus be quite useful.

6 Conclusion and Future Work

In this paper we presented our experience with modeling and checking linearizability of complex concurrent data structure algorithms. This is the first work that describes in detail the challenges and choices that arise in this process. We believe our experience will be instructive to anyone who attempts to specify and model check complex concurrent algorithms.

Based on our experience, we suggest several directions for enhancing the SPIN model checker to deal with heap manipulating concurrent algorithms such as concurrent sets. The two critical features include garbage collection support and the symmetry reduction optimization. Further, we proposed several items for future work in analysis and verification, including the automated inference of linearization points.

References

1. Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
2. Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.
3. Dragan Bosnacki, Dennis Dams, and Leszek Holenderski. Verifying commit-atomicity using model-checking. In *SPIN*, 2000.
4. Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV*, 2006.
5. Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. *SIGPLAN Not.*, 42(6):12–21, 2007.
6. Claudio Demartini, Radu Iosif, and Riccardo Sisto. dspin: A dynamic extension of spin. In *SPIN*, pages 261–276, 1999.
7. Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, pages 27–37, 2005.
8. Cormac Flanagan. Verifying commit-atomicity using model-checking. In *SPIN*, 2004.
9. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proc. of conf. On Principles Of Distributed Systems (OPODIS 2005)*, pages 3–16, 2005.
10. Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, Bill Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.
11. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *Trans. on Prog. Lang. and Syst.*, 12(3), 1990.
12. Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, pages 254–261, 2001.
13. Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In *SPIN*, pages 20–33, 2000.
14. Noam Rinetzky Ivana Mijajlovic, Peter O'Hearn and Hongseok Yang. Abstraction for concurrent objects. In *ESOP'09*, 2009.
15. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.
16. Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
17. Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.
18. Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPOPP*, pages 45–54, 2009.
19. Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.
20. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
21. Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, pages 335–348, 2009.
22. Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.
23. Jeannette M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2):164–182, 1993.