Deriving Linearizable Fine-Grained Concurrent Objects

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Abstract

Practical and efficient algorithms for concurrent data structures are difficult to construct and modify. Algorithms in the literature are often optimized for a specific setting, making it hard to separate the algorithmic insights from implementation details. The goal of this work is to systematically construct algorithms for a concurrent data structure starting from its sequential implementation. Towards that goal, we follow a construction process that combines manual steps corresponding to high-level insights with automatic exploration of implementation details. To assist us in this process, we built a new tool called PARAGLIDER. The tool quickly explores large spaces of algorithms and uses bounded model checking to check linearizability of algorithms.

Starting from a sequential implementation and assisted by the tool, we present the steps that we used to derive various highly-concurrent algorithms. Among these algorithms is a new fine-grained set data structure that provides a wait-free `contains` operation, and uses only the compare-and-swap (CAS) primitive for synchronization.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; D.2.4 [*Program Verification*]

General Terms Algorithms, Verification

Keywords concurrent algorithms, verification, synthesis, model checking, linearizability

1. Introduction

Concurrent data-structures, also known as *concurrent objects* [15], allow the programmer of a concurrent system to work with the illusion of a sequential data-structure, while permitting a high-level of concurrency. To achieve this objective, concurrent objects are usually implemented using fine-grained locking, or other fine-grained synchronization mechanisms such as compare-and-swap (CAS). Unfortunately, this makes them notoriously hard to design, implement, and verify. This is especially true when their implementations employ low level pointer manipulations (see, e.g., [8]).

Universal construction methodologies such as [14], provide a systematic way to construct concurrent objects, but produce inefficient results. Other popular concurrency mechanisms such as software transactional memory [11], if used naively, can lead to inefficiencies due to false sharing: unaware of the underlying data structure invariants, a transactional operation may restart unnecessary even if its effect is benign. To gain efficiency, one can try to reduce the scope of the transaction, but then we are faced with similar challenges as designing fine-grained data structures.

Direct construction of specific algorithms is the realm of experts, and produces algorithms that are very efficient, but are already specialized for a particular environment. This process mixes algorithmic insights with implementation complexity, and makes it hard to adapt the algorithms to a different setting, modify them, and verify their correctness.

In this paper, we show how to systematically derive fine-grained concurrent objects, starting from a sequential implementation. Our derivation process combines manual steps that correspond to high-level insights with automatic exploration of implementation details. The exploration procedure, implemented in a tool called PARAGLIDER, helps the designer focus attention on specific algorithmic variations by quickly rejecting incorrect ones. We show how the systematic derivation, assisted by the tool, yields interesting and practical concurrent algorithms.

To guarantee that a concurrent object appears to the programmer as a sequential data-structure, concurrent objects are often required to be *linearizable* [15]. Intuitively, linearizability provides the illusion that any operation performed on a concurrent object takes effect instantaneously at some point between its invocation and its response. Existing approaches for automatic verification and checking of linearizability (e.g., [9, 10, 2, 30]) and of related correctness conditions (e.g., [6, 5]) are valuable. However, these generally apply when the concurrent object has already been implemented. In this paper, we follow a different direction, and provide a tool that can assist a designer in a systematic derivation process of linearizable fine-grained concurrent objects.

1.1 Motivating Example

Fig. 1 shows a standard sequential implementation of a set data structure. This implementation uses an underlying singly linked-list. The list is sorted in ascending key order, and uses two sentinel nodes *head* (with the smallest possible key) and *tail* (with the largest possible key). The set supports three operations: `add`, `remove`, and `contains`, with their standard meaning. All operations use a macro LOCATE to traverse the list and locate an item based on the value of its key.

Fig. 2 shows a concurrent set algorithm derived by a designer using PARAGLIDER. We explain the exact details of this algorithm in Sec. 4.3. For now, it suffices to note that this algorithm is quite distant from the sequential implementation of Fig. 1, and in particular, uses the lower-level CAS synchronization primitive. Note that designing practical algorithms that use a double compare and swap primitive (DCAS) is a challenging task [8]. Algorithms relying only on CAS are even more complex and involved [21].

The design of a concurrent object requires expertise and designer insights, but also requires dealing with a large number of implementation details. Particular design decisions are being made

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

```
bool add(int key) {
 Entry *pred,*curr,*entry;
   LOCATE(pred, curr, key)
   k=(curr->key == key)
   if(k) return false
   entry = new Entry(key)
   entry->next = curr
   pred->next = entry
   return true
}
```

```
bool remove(int key) {
 Entry *pred,*curr,*r;
   LOCATE(pred, curr, key)
   k=(curr->key ≠ key)
   if(k) return false
   r = curr->next
   pred->next = r
   return true
}
```

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

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

Figure 1. A sequential implementation of a set algorithm based on a sorted singly-linked-list.

```
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
  val=CAS(&pred->next, ⟨curr.ptr, 0⟩, ⟨entry.ptr, 0⟩)
  if(¬val) goto restart
  return true
}
```

```
boolean remove(int key) {
 Entry *pred,*curr,*r
restart:
  LOCATE(pred, curr, key)
  k=(curr->key ≠ key)
  if(k) return false
  r = curr->next
  lval=CAS(&curr->next, r, ⟨r.ptr, 1⟩)
  if(¬lval) goto restart
  pval=CAS(&pred->next, ⟨curr.ptr, 0⟩, ⟨r.ptr, 0⟩)
  if(¬pval) goto restart
  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
}
```

Figure 2. A concurrent set algorithm using a marked bit to mark deleted nodes. The bit is stored with the *next* pointer in a single word. Synchronization is implemented using CAS. `contains` does not use synchronization and does not restart. The LOCATE is the one of Fig. 1.

based on the environment (e.g., available memory model and synchronization primitives) and the requirements from the algorithm (e.g., required memory overhead and progress guarantees). Each design decision entails different low level implementation details. The designer wishing to obtain a working algorithm for her specific setting is often forced to follow a single path of design choices due to the cost of exploring alternatives.

When designing such algorithms, the high-level skeletal structure of the algorithm is often known. The key to an efficient design process is the ability to quickly check various design choices. The designer may be facing questions relating to alternative design choices such as: can synchronization be reduced by adding more coordination meta-data? can I obtain a correct algorithm with lower space overhead? can I obtain a similar algorithm for a slightly different environment? (e.g., assuming no garbage collection.)

Our system is geared towards helping such experts in the design process of concurrent objects. The system allows the domain expert to specify her *insights* and assists her by exploring a space of algorithms based on the provided insights.

For example, the "insight" in the concurrent algorithm of Fig. 2 is that a *marked bit* has to be used to coordinate between threads. Introducing a marked bit for a deleted node was first used in the lock-free FIFO queue algorithm of Prakash et. al. [23] . This insight, later used in [12], can lead to many different implementations, depending on the environment of the concurrent object, and requirements imposed on it. For example, Michael [21] used this insight as a basis for a concurrent set algorithm that operates with explicit memory management (making the problem significantly harder). Heller et. al. [13] used this insight as the basis of a lock-based algorithm. In this paper, we show how PARAGLIDER can assist a designer in the systematic derivation of such algorithms.

1.2 Domain-specific Exploration

The usage scenario of PARAGLIDER is one that combines *manual transformations* with *automatic exploration*. The designer provides the insightful leaps by manual transformations of algorithm schemas, and the system fills the lower-level details by performing a combinatorial search around each of the provided schemas. Since different algorithms produced from the same schema can have different tradeoffs in terms of space overhead, synchronization and progress, PARAGLIDER produces a set of results. The designer can choose the appropriate algorithm from the reported set.

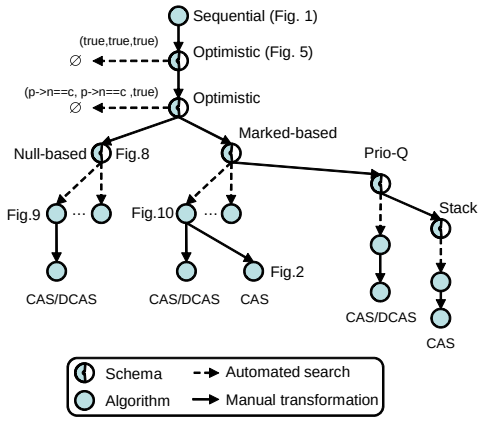

Figure 3. Flow of derivation in this paper.

PARAGLIDER allows the designer to provide domain-specific knowledge about the space of algorithms being explored by: (i) defining a partial order of correctness-implication between algorithms; (ii) defining an equivalence between algorithms. This enables the exploration procedure to often infer the correctness (or incorrectness) of an algorithm by the correctness (or incorrectness) of another algorithm in the search space. This reduces the number of algorithms that are checked by the model checker to less than 3% of the total number of algorithms in the space. This reduction is the key to feasibility of exploration.

1.3 Derivation Flow and Correctness Conditions

Fig. 3 shows an overview of the example derivation process described in this paper. The starting point is the sequential algorithm of Fig. 1. This algorithm, when considered in a concurrent setting, is not linearizable. A naive solution would be to put each operation

inside a coarse-grained atomic section. In this case, the resulting algorithm would be linearizable, but permit no concurrent operations. The basic question that we ask is:

> *How do we derive an algorithm with fine-grained synchronization, while preserving correctness?*

We answer this question by following a systematic derivation process which combines manual transformations (shown as solid arrows), with automatic exploration (shown as dashed arrows). The manual transformations capture designer insights and produce algorithm schemas (shown as half-full circles) from which automatic exploration can proceed. Automatic exploration instantiates a schema into a set of algorithms (each shown as a circle) by filling the missing lower-level implementation details (such as order between operations, and partition to atomic sections). We use $\emptyset$ to denote cases when all instances of a schema fail the correctness check.

Correctness Conditions All automatic exploration is done while checking for linearizability. More details are provided in Sec. 5.

1.4 Main Results

The main contributions of this paper are:

- PARAGLIDER, a tool that assists algorithm-designers to systematically derive concurrent algorithms by exploring an algorithm space based on the designers insights.
- The tool checks algorithms in the space for *linearizability*, and supports checking of linearizability both with fixed linearization points and with automatic linearization.
- The tool uses a domain-specific exploration that leverages the relationship between algorithms in the space to reduce the number of algorithms that have to be checked by the model checker.
- We show how the tool is used in a systematic derivation process. We derive a variety of concurrent set algorithms, including algorithms close to the ones of [12], [13] and [21]. Specifically, we derive a new concurrent set algorithm that provides a wait-free `contains` operation, and uses only the compare-and-swap (CAS) primitive for synchronization.

1.5 Assumptions

Our current implementation makes the following assumptions:

- we assume an underlying sequentially consistent memory model. Our tool can operate without this assumption, but the correctness condition will have to be adapted to that setting (e.g., sequential consistency, as in [5]), and the state space explored by the checking procedure will increase significantly.
- we assume that algorithms operate in the presence of a garbage collector, and that a node removed from a data-structure is never reused. This is a standard assumption (e.g., [13]) that allows us to focus on the algorithms and ignore details of explicit memory management. The addition of explicit memory management is a separate challenging problem (see, e.g., [21]).

We focus on the exploration of large spaces of algorithms and use *bounded model-checking* to check for *linearizability* and safety of algorithms. Automatic verification of linearizability for the algorithms we explore remains a problem for future work.

At this point, the system can be viewed as producing a set of candidate algorithms in a systematic manner. In particular, the system narrows the space of algorithms that the user has to examine by ruling out algorithms observed as incorrect.

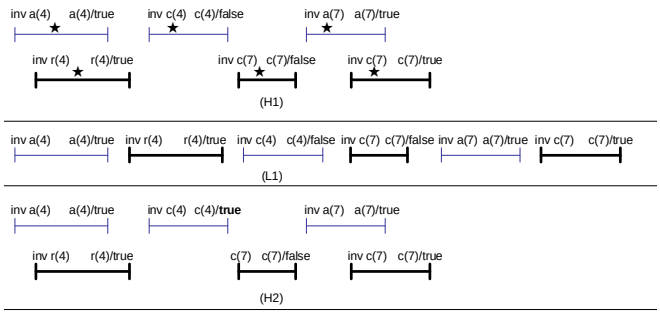

Figure 4. Concurrent histories and possible sequential histories corresponding to them.

2. Background: Linearizability

Linearizability [15] is a commonly used correctness criterion for implementations of concurrent objects. Intuitively, linearizability provides the illusion that any operation performed on a concurrent object takes effect instantaneously at some point between its invocation and its response.

The linearizability of a concurrent object is verified 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* [17], and *serializability* [22] also require that a concurrent history be equivalent to some sequential history in which operations appear to have executed atomically (i.e., without interleaving). However, these criteria differ on the requirements on ordering of operations. Sequential consistency requires that operations in the sequential history appear in an order that is consistent with the order seen at individual threads. Serializability is defined in terms of transactions, and requires that transactions appear sequentially, that is, without interleavings. Note that a transaction may include several operations, and may operate on several objects.

Compared to these correctness criteria, linearizability is more intuitive, as it preserves the real-time ordering of non-overlapping operations. In addition, linearizability is *compositional*, meaning that a system consisting of linearizable objects is guaranteed to be linearizable [15].

EXAMPLE 2.1. *Fig. 4 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.

Checking linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution.

In some cases, it is possible to specify for each operation a point in the code (more generally, a set of alternative points) in which the operation appears to take place instantaneously. Such points are commonly referred to as linearization points. When the linearization points of a concurrent object are known (e.g., by user specification), they induce an order between overlapping operations of a concurrent history. This obviates the need to enumerate all possible permutation for finding a linearization.

EXAMPLE 2.2. *Consider the history* H_1 *of Fig. 4, the star symbols in the figure denote the occurrence of a user-specified linearization point in each operation. Using 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.

We can check linearizability in two settings: (i) *automatic linearization*—the system explores all permitted permutations of a concurrent history to find a valid linearization; (ii) *fixed linearization points*—the linearization point of each operation is defined in terms of a user-specified statement in the program in which it appears to take place. The ability to check algorithms using automatic linearization is essential for the exploration process, as the linearization points of algorithms being explored are unknown. In this paper, we focus on checking linearizability, as it is the appropriate condition for the domain of concurrent objects [15]. Our tool can also use other conditions (e.g., operation-level serializability).

3. Paraglider

In this section, we show how our domain specific exploration procedure, implemented in a tool called PARAGLIDER, is able to efficiently explore a vast number of combinations with a small number of invocations of the checking procedure. For now, we treat the checking procedure as a black box (see Sec. 5 for details).

3.1 Input / Output

The exploration framework uses the following inputs:

- a *program schema* with *placeholders* for missing code-fragments.
- a (correct) sequential implementation of the concurrent object, used as an executable specification of correct behavior.
- optional: a specification of linearization points.

The output of the exploration procedure is a set of instantiated schemas (referred to as "algorithms"), that have been checked by the checking procedure.

Informally, a placeholder defines a set of code fragments that can be used to fill the missing part in the program schema. A placeholder is defined over a set of *building blocks*, and a set of constraints on how these blocks can be used. The building blocks can be viewed as the statements of a domain specific language for constructing concurrent data structures. Generally, any building block that references a shared location contains one memory access (although it may operate on any number of local values). For example, a building block $r = curr{->}next$ reads the $next$ pointer of $curr$ into a local variable r. We present building blocks for the algorithms explored in this paper in Section 4.

We allow the user to leave the exact details of a building block undetermined, and let a block be parametric on the expressions it is using. The set of possible expressions used by a building block can be defined using a regular expression (we assume a predetermined bounded length). For example, the parametric block $x = y.[f|g|h]$ can be instantiated to the three *instantiated blocks* $x = y.f$, $x = y.g$ or $x = y.h$. The exploration preformed by PARAGLIDER will try each of these instantiated blocks.

In this paper, a placeholder can be constrained by *sequencing constraints* that restrict the permitted sequences of blocks, and *atomicity constraints* that specify which blocks have to be executed atomically.

DEFINITION 3.1. *A placeholder* ph *is a pair* $\langle B, C\rangle$*, where* B *is a set of building blocks, and* C *is a set of atomicity constraints and sequencing constraints expressed as regular expressions over* B*. For atomicity constraints, we write* $[b_1, b_2]$ *when a block* b_1 *must appear in the same atomic section with* b_2 *in* ph *(note that this does not restrict the order in which they appear within the atomic section). For sequencing constraints, we use the following shorthand notations: we write* $b_1 < b_2$ *when* b_1 *must appear before* b_2 *in* ph*, and* (b_1, b_2) *when* b_1 *must appear adjacent to* b_2 *in* ph.

A placeholder instance is a single code fragment (sequence of instantiated blocks and atomic sections) that satisfies the constraints of the placeholder.

DEFINITION 3.2. *Given a placeholder* $ph = \langle B, C\rangle$*, a placeholder instance is a code fragment* cf *such that all blocks of* cf *are instantiated from* B*, and* cf *satisfies the constraints specified in* C*. In particular, the code fragment* cf *includes an equivalence relation* $atom(cf) \subseteq B \times B$*, that satisfies the atomicity constraints. We note that* $atom(cf)$ *partitions* cf *to atomic sections. Given a placeholder* ph*, we use* D_{ph} *to denote the set of all placeholder instances for* ph*, and refer to it as the* domain *of* ph.

A program schema consists of a skeleton and a set of placeholders (including expression placeholders). Given a schema S, we use $PH(S)$ to denote the set of placeholders in S.

Given a program schema S, a *placeholder assignment* A is a (partial) function assigning a placeholder to a specific instance in its domain. When A assigns an instance for each placeholder of S, we say that A is a *complete assignment*.

A *schema instance* is a pair of a schema S, and a complete assignment A, assigning a placeholder instance for each placeholder of S, i.e., a schema in which all placeholders are instantiated.

3.2 Exploration

Our system exhaustively explores the space of placeholder assignments. For most problems, this space is huge and contains millions of potential assignments. A key ingredient of PARAGLIDER is the ability to reduce this space by exploiting domain specific knowledge. PARAGLIDER leverages a partial order of implied correctness defined between assignments, reducing the number of algorithms that are checked to less than 3% of the total number of algorithms in the space. This reduction is the key to feasibility of exploration.

Given a placeholder ph we allow the user to define a partial order between placeholder instances. This partial order defines implied correctness between assignments, thus allowing the search to use the correctness (or incorrectness) of an assignment to determine the correctness (or incorrectness) of other assignments.

The system is pre-equipped with a partial order defined between instances based on atomic sections. Given a placeholder ph and two placeholder instances $i_1, i_2 \in D_{ph}$, we say that $i_1 \leq i_2$ when for every two (instantiated) blocks b_1, b_2, they appear in the same order in i_1 and i_2, and if $(b_1, b_2) \in atom(i_1)$ then $(b_1, b_2) \in atom(i_2)$.

This partial order captures the domain-specific knowledge that if a given code-fragment yields a correct algorithm, then increasing the scope of atomic sections in this code fragment maintains correctness. In the search, our system also leverages this constraint in the opposite direction — when a code fragment yields an incorrect algorithm, any code fragment that is less atomic will also yield an

incorrect algorithm. We note that one has to be careful when matching the correctness criterion used by the checking procedure with the notion of implied correctness.

DEFINITION 3.3 (Ordering schema instances). *Given a schema S and two complete assignments A_1 and A_2, we say that A_1 implies the correctness of A_2, and write $A_1 \leq A_2$ when for every $ph \in PH(S)$, $A_1(ph) \leq A_2(ph)$.*

The system also allows a user to define a notion of equivalence between placeholder instances. The system is pre-equipped with an equivalence of instances based on atomic sections. Intuitively, when the operations inside an atomic section are independent of each other in terms of dataflow, they can be reordered. Two placeholder assignments that only differ on the ordering of independent operations inside an atomic section are considered equivalent.

The system uses implied correctness and incorrectness based on schema instance ordering and equivalence to reduce the number of algorithms that have to be checked. In Section 4.5, we report experimental results that show the effectiveness of this reduction.

Correctness In this paper we focus on checking linearizability, but our tool can use any other checking/verification procedure.

4. Concurrent Set Algorithms

In this section, we demonstrate how a designer uses the system to construct several fine-grained data structures starting from a sequential implementation. The section follows the derivation roadmap shown in Fig. 3.

We explore algorithms by initially considering the general notion of an atomic section, without specifying how it is implemented. As a second step, we show how to realize the atomic sections in the derived algorithms via the CAS and DCAS synchronization primitives. Keeping the first part of the derivation at the level of atomic sections permits alternative realizations (e.g., via locks).

4.1 From Sequential to Optimistic Concurrency

Our first observation is that the traversal of the list searching for the appropriate key (in LOCATE) is only reading values from the list, and not modifying it. In the absence of concurrent modifications, the search part of an operation could be performed concurrently. This motivates our first step: using optimistic concurrency.

4.1.1 Optimistic Concurrency

We apply a well-known concurrency transformation due to [16] and obtain a basic schema using optimistic concurrency. In this method, the optimistic list traversal no longer uses synchronization. However, after the traversal, we atomically check a *validation* condition and perform the operation if the condition holds. The resulting schema is shown in Fig. 5, where the validation conditions are left as unspecified expression placeholders $\langle$VALADD$\rangle$, $\langle$VALREM$\rangle$, $\langle$VALCT$\rangle$. In order to turn the schema of Fig. 5 into a correct (but still a coarse-grained) algorithm, we need to find a suitable assignment of validation conditions.

While we can also find the conditions via a blind search over some set of user-provided expressions, it may be valuable for a designer to observe the states that cause failure and consider a number of alternatives. The designer can try and construct a validation condition by starting with the weakest ($true$), or by starting with the strongest ($false$) condition. Starting with the weakest and strengthening it seems more natural, as it may expose violations of a safety property that would help us identify how to strengthen the condition. Starting with the strongest condition and weakening it is more challenging, as the base case is a one of non-termination (the algorithm always restarts).

```
boolean add(int key) {
 Entry *pred,*curr,*entry;
restart:
 LOCATE(pred, curr, key)
 atomic
 [if(⟨VALADD⟩){
  | k=(curr->key == key)
  | if(k) return false
  | entry = new Entry(key)
  | entry->next = curr
  | pred->next = entry
  | return true
  |}
 goto restart
 }

boolean contains(int key) {
 Entry *pred,*curr;
restart:
 LOCATE(pred, curr, key)
 atomic
 [if(⟨VALCT⟩){
  | k=(curr->key==key)
  | if(k) return true
  | if(¬k) return false
  |}
 goto restart
}

boolean remove(int key) {
 Entry *pred,*curr,*r;
restart:
 LOCATE(pred, curr, key)
 atomic
 [if(⟨VALREM⟩){
  | k=(curr->key ≠ key)
  | if(k) return false
  | r = curr->next
  | pred->next = r
  | return true
  |}
 goto restart
}
```

Figure 5. An algorithm schema using optimistic concurrency with $\langle$VALADD$\rangle$,$\langle$VALRM$\rangle$,$\langle$VALCT$\rangle$ as unknown validation conditions.

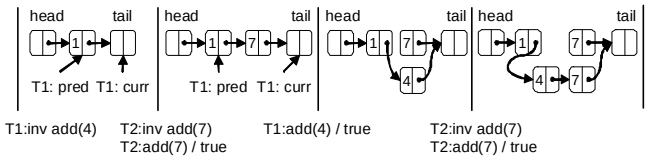

Figure 6. Invalid execution with the validation condition *true*.

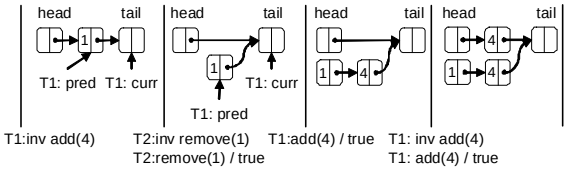

Figure 7. An invalid execution with the validation condition $pred{-}{>}next{=}curr$.

4.1.2 Finding Validation Conditions

We choose to start with the weakest conditions mainly because the underlying checking procedure provides a counterexample when a safety property is violated.

We start by setting $\langle$VALADD$\rangle$, $\langle$VALCT$\rangle$ and $\langle$VALREM$\rangle$ assigned to *true*. We then run the system with only `add` operations. Fig. 6 shows an invalid execution under this choice. Thread T_1 invokes `add(4)`, finds the location in the list, and stops. At this state, the pointer $pred$ of T_1 (hereafter T_1: $pred$) points to the node 1, and T_1: $curr$ points to the tail of the list. Before T_1 continues, a thread T_2 invokes and completes the operation `add(7)`. This results in the node 7 inserted between T_1: $pred$ and T_1: $curr$. Next, T_1 continues execution of `add(4)`, resulting in the loss of the node 7. Finally, another invocation of `add(7)` by T_2 adds 7 to the list, and returns $true$ where all corresponding sequential executions return $false$. At this point, the system reports a linearizability violation.

```
boolean add(int key) {
 Entry *pred,*curr,*entry
restart:
 LOCATE(pred, curr, key)
 ⟨ADD⟩
 return true
}

⟨ADD⟩ = {
 A1: val=(pred->next==curr)
 A2: k=(curr->key == key)
 A3: if(k) return false
 A4: pred->next = entry
 A5: entry = new Entry(key)
 A6: entry->next = curr
 A7: if(¬val) goto restart
} with {
 (A1, A7), (A2, A3),
 A3 < A4, A7 < A4
}
```

```
boolean remove(int key) {
 Entry *pred,*curr,*r
restart:
 LOCATE(pred, curr, key)
 ⟨REMOVE⟩
 return true
}

⟨REMOVE⟩ = {
 R1: val=(pred->next==curr)
 R2: k=(curr->key ≠ key)
 R3: if(k) return false
 R4: r = curr->next
 R5: curr->next = null
 R6: pred->next = r
 R7: if(¬val) goto restart
} with {
 (R1, R7), (R2, R3)
 R3 < R6, R7 < R6, R7 < R5
}
```

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

LOCATE(pred, curr, key) {
  pred=head
  curr = head->next
  while(curr->key < key){
   pred=curr
   curr = curr->next
   if(curr==null) goto restart
  }
}
```

Figure 8. Paraglider specification for a concurrent set algorithm using $curr{->}next = null$ for deleted nodes.

The cause of the problem is that an update done by `add` assumes $pred{->}next == curr$, a property that can be violated by another thread between the search and the update. To prevent this scenario, we manually strengthen ⟨VALADD⟩ to check whether $pred{->}next$ still points to $curr$. When that is the case, `add` can proceed, otherwise the operation is restarted. We again check the algorithm running the system only with `add` operations, and no error is reported. We therefore move on to derive a validation condition in the presence of `remove` operations. A similar counter-example between `remove` operations leads to strengthening ⟨VALREM⟩ to $pred{->}next == curr$. Unfortunately, the resulting algorithm is still incorrect. The problem is due to interference between `add` and `remove`. Fig. 7 shows an example of an invalid execution. First, thread T_1 invokes `add(4)`, finds the location in the list, and stops. Thread T_2 then executes a full `remove(1)` operation and removes the node pointed to by T_1: $pred$ from the list. Next, T_1 resumes and inserts key 4 between $pred$ and $curr$. Because the node pointed to by $pred$ was already removed from the list, the update will cause node 4, inserted by T_1, to be lost. Finally, another invocation of `add(4)` by T_1 adds 4 to the list and returns $true$, thus observing a violation of linearizability (as all sequential executions would have returned $false$).

This sample execution suggests the need for a mechanism to *prevent removed nodes from being used in other operations*. At this point, it is useful to stop and ask the question:

> *How can we systematically strengthen the validation condition based on this information?*

In principle, we follow two general guidelines: (i) perform more work without adding space for coordination metadata, (ii) add space for coordination metadata and more work based on that metadata.

4.2 Performing Additional Work

The high-level idea is to make the fact that a node has been removed observable to threads that may be trying to use it. Our insight is that removal of a node can be made observable to other threads by setting the `next` field of the node to $null$ when it is being removed. Since this might break the traversal performed by the optimistic search, we have to adjust LOCATE, in addition to the operations' building blocks to correctly act upon a pointer that was set to $null$. Fig. 8 shows the schema for algorithms using the above insight. It consists of an implementation of LOCATE and `contains`, and skeletal implementations for `add` and `remove`. The LOCATE macro is manually adapted from the sequential implementation, adding a restart when $curr == null$.

```
ADD = {
k=(curr->key == key)
if(k) return false
entry = new Entry(key)
entry->next = curr
atomic
[ val=(pred->next==curr)
  if(¬val) goto restart
  pred->next = entry
]
}
```

```
REMOVE = {
k=(curr->key ≠ key)
if(k) return false
atomic
[ val=(pred->next==curr)
  if(¬val) goto restart
  r = curr->next
  curr->next = null
  pred->next = r
]
}
```

Figure 9. A placeholder assignment for the schema of Fig. 8 resulting in a *new* concurrent set algorithm.

The skeletal implementation for `add` uses the placeholder ⟨ADD⟩, and that of `remove` uses ⟨REMOVE⟩. Building blocks are shown as labeled statements inside the placeholder definitions. Note that most blocks correspond to the same core actions performed by the original sequential implementation. The block $R5$ reflects the designer insight that the next pointer of a removed node should be set to $null$. Blocks $A1$ and $R1$ correspond to the validation expressions we obtained in the previous section.

The placeholder ⟨ADD⟩ uses two constraints clumping together certain building blocks, and two constraints that force an order between blocks. The constraint $(A1, A7)$ forces the conditional restart to be placed adjacent to the computation of the validation condition (the order between the blocks is forced by dataflow constraint that is automatically determined by the system). The constraint $(A2, A3)$ forces the conditional return of $false$ to be adjacent to the computation of its condition (computation of k). The constraints $A3 < A4$ and $A7 < A4$ force an order in which the operation checks the conditions on key equality and the validation condition before applying the update. These constraints are added in order to reduce the size of the search space, and are based on designer knowledge. In addition to user-specified constraints, the system automatically generates dataflow constraints, and adds these as ordering constraints between building blocks.

While the insight of using $null$ is known to the designer, the exact way in which this should be implemented is unclear. In particular, the designer would like to obtain the least atomic algorithm that can be constructed using this insight. This is where the system assists the process, by systematically exploring the space of algorithms that can be constructed from the specified schema. For this schema, PARAGLIDER found 16 instances. Fig. 9 shows one of the algorithms produced by the system, other algorithms found are variations of this algorithm. Note that there are still many possible combinations that may yield different algorithmic variations.

In particular, there are algorithms where the validation check and restart take place before the check of key. In addition, note that finding a single correct instance for a schema is not enough. We are interested in investigating the tradeoffs between multiple correct instances of a schema.

Next, we show how the atomic sections in this algorithm can be directly expressed using low-level synchronization primitives.

4.2.1 Using Low-level Synchronization Primitives

The atomic section in the `add` operation contains a single read, write and comparison operations. This atomic section can be directly implemented as a CAS. The CAS operation compares the content of a given address to an expected value, and if the value matches, atomically writes a new value to the given location. The precise meaning is:

$$
\begin{array}{l}
CAS(addr, old, new)\ \{ \\
\ \textbf{atomic} \\
\ \left[\begin{array}{l} if((*addr == old)\ \{*addr = new;\ return\ true\} \\ \ else\ return\ false \end{array}\right. \\
\}
\end{array}
$$

The atomic section in the `add` is:

$$
\begin{array}{l}
\textbf{atomic} \\
\left[\begin{array}{l} val = (pred{->}next == curr) \\ if(\neg val)\ goto\ restart \\ pred{->}next = entry \end{array}\right.
\end{array}
$$

Since the update of $pred{->}next$ is only performed conditionally, and val is a local value computed only once, the goto restart block can be taken outside the atomic.

$$val = CAS(\&pred{->}next,\quad curr, entry)$$

The atomic section in the `remove` operation contains two reads, one comparison, and two writes. We would like to implement this atomic section using a DCAS. Similarly to the `add`, we note that we can move the local operation that is the restarting block after the atomic section. The DCAS operation compares and swaps two locations atomically:

$$
\begin{array}{l}
DCAS(addr1, addr2, old1, old2, new1, new2)\ \{ \\
\ \textbf{atomic} \\
\ \left[\begin{array}{l} if((*addr1 == old1)\&\&(*addr2 == old2))\{ \\ \ *addr1 = new1;\ *addr2 = new2;\ return\ true \\ \} else\ return\ false \end{array}\right. \\
\}
\end{array}
$$

However, in order to use a DCAS, we first need to transform it to perform $r = curr{->}next$ outside of the atomic section, so it matches the meaning of a DCAS. We use a standard optimistic concurrency transformation in which an atomic read of a value is replaced by a non-atomic read followed by a validating comparison. Using this transformation, we rewrite the atomic block as:

$$
\begin{array}{l}
r = curr{->}next \\
\textbf{atomic} \\
\left[\begin{array}{l} val = (pred{->}next == curr) \wedge (curr{->}next == r) \\ curr{->}next = null \\ pred{->}next = r \end{array}\right.
\end{array}
$$

The atomic section can now be transformed to:

$$val = DCAS(\&pred{->}next,\quad \&curr{->}next, curr, r, r, null)$$

4.3 Adding Coordination Metadata

The Null-based algorithms obtained in Section 4.2 were designed under the assumption that no additional space should be used for coordination metadate. The best algorithm we obtained had two main disadvantages. First, the optimistic search in an operation might be disrupted and forced to restart when the `next` pointer of a node has been set to null. Secondly, our goal is to reach

```
boolean remove(int key) {
 Entry *pred,*curr,*r
restart:
 LOCATE(pred, curr, key)
 ⟨REMOVE⟩
 return true
}

 ⟨REMOVE⟩ = {
  R1 : val=(pred->next==curr) (∧mp)? (∧mc)?
  R2 : k=(curr->key ≠ key)
  R3 : if(k) return false
  R4 : r = curr->next
  R5 : curr->marked = true
  R6 : pred->next = r
  R7 : if(¬val) goto restart
  R8 : mp = ¬pred->marked
  R9 : mc = ¬curr->marked
 } with {
  (R2, R3), (R1, R7)
  R3 < R6, R7 < R6
 }
```

Figure 10. Paraglider specification for a `remove` operation using a marked bit for deleted nodes.The block $R1$ is a parametric block.

algorithms that only use CAS, rather than DCAS as the best null-based algorithm does.

From our previous study on another class of concurrent algorithms [29], we know that it is possible to trade-off space for synchronization. That is, introducing more space may allows us to reduce the required synchronization. To that end, we introduce a separate marked field in each node denoting that the node is being removed. This is inline with modern algorithms who interpret the setting of that bit as logical removal of the node.

Adding a Marked Bit We modify the schema to include a separate marked bit to denote deleted nodes. Fig. 10 shows the schema for the `remove` operation using a marked bit. We omit the rest of the schema since it is similar to the schema of Fig. 8.

In the new schema, we modify the LOCATE code to remove the (redundant) restart on the case that `curr==null`. We modify the building blocks of Fig. 8 by: (i) replacing the block $R5$ with a new block setting a marked bit; (ii) adding new blocks for reading the marked bit of $pred$ and $curr$ (blocks $R8$ and $R9$ for `remove`, and similar blocks for `add`); (iii) replacing the blocks computing the validate condition $A1$,$R1$ by new validate conditions as described below.

After the addition of the marked bit, it is no longer clear what the validation conditions should be. We know from our experience with the previous algorithm that it must contain the check for $pred{->}next == curr$. However, we do not know the marked bits of which nodes should be checked in each condition. We therefore define the building blocks computing the conditions as parametric blocks. A parametric block can draw its value from a set of values, specified as a regular expression. PARAGLIDER then searches through these value assignments exhaustively as part of the exploration. In the case of the validate condition, we define three parametric blocks, one for each operation. The parametric blocks for all three operations are of the form:

$$val = (pred{->}next == curr)\ (\wedge mp)?\ (\wedge mc)?$$

It is important to note that knowing the meaning of the expression, the fact that it is a *validate* expression, allows us to impose an order on the search. For a given ordering of the blocks, and a given assignment of atomic sections, if an expression e yields an incorrect algorithm, then any weaker expression also yields an incorrect algorithm (as it permits a superset of values permitted by e).

Running the system with the new set of blocks produces 6 algorithms. Note that these algorithms only check the $marked$ bit of the $pred$ node (and not the $curr$ node). In particular, the algorithm of Fig. 11 is one of the results with the smallest atomic

```
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
}

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

```
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
}
```

Figure 11. A set algorithm using a marked bit to mark deleted nodes. A variation of [13] that uses a weaker validation condition.

sections. As we will see in the next section, this can be leveraged towards an efficient implementation of the algorithm. One of the differences between the algorithm in Fig. 11 and the one presented by Heller et. al. [13] is that it uses weaker validation conditions. The algorithm of [13] also checks the $marked$ bit of the node $curr$.

At this point in the derivation, we have obtained all algorithms that can be produced from the schema using a marked bit.

4.3.1 Using Low-level Synchronization Primitives

After obtaining several fine-grained algorithms using generic atomic sections, our goal was to bring them closer to practical implementations. The algorithm of Fig. 11 uses atomic sections that are similar to the ones we saw in the algorithm of Fig. 9. However, the algorithm reads the content of an additional memory location (the marked bit) inside the atomic sections of `add` and `remove`. To implement this algorithm using CAS/DCAS we follow [12, 21] in which the marked bit is stored together with the pointer in the same atomic storage unit (ASU). Using a single ASU (usually a machine word) to record both the next pointer and the marked bit of a node makes it possible to read both, atomically, with a single read operation.[1] We demonstrate the use of a single ASU on the atomic section of the `add` operation. We use $\langle ptr, bit\rangle$ to denote a pair of a reference and a bit value stored in the same ASU. The atomic section of the `add` operation can be rewritten as:

```
atomic
[ ⟨pnext, pmrk⟩ = pred->next
| val = (pnext == curr) ∧ ¬pmrk
| if (val) pred->next = entry
```

This, in turn, can be now expressed as a CAS, similar to the CAS in Section 4.2.1:

$$val = CAS(\&pred\text{->}next, \langle curr.ptr, 0\rangle, \langle entry.ptr, 0\rangle)$$

where $curr.ptr$ refers to the pointer component of the pointer $curr$, and $\langle curr.ptr, 0\rangle$ is the composite pointer with 0 as the value of the marked bit. Similarly for $entry$.

The atomic section of the `remove` operation can be rewritten to use only CAS operations. To do that, we follow the same basic idea as outlined in Section 4.2.1: a standard optimistic concurrency transformation in which an atomic read of a value is replaced by a non-atomic read followed by a validating comparison. We first read $curr\text{->}next$ into a temporary and use it in a CAS in order to set the marked bit of $curr$. If the CAS fails, we restart the operation.

Since the marked bit is stored together with the next pointer, writing the marked bit requires an atomic manipulation of the entire `next` field. As a result, the marking operation itself, although non-conditional, now requires a CAS. This is purely due to the implementation choice (space reduction) of storing the marked bit together with the pointer. The marking is therefore performed by:

$$r = curr\text{->}next$$
$$lval = CAS(\&curr\text{->}next, r, \langle r.ptr, 1\rangle)$$

where the value read into r is also used later for the removal from the list. Similarly, we use another CAS to check the validate of the actual removal, and to perform it atomically. The process results in the least atomic algorithm we managed to derive and was already shown in Fig. 2.

4.4 Adapting Set Algorithms

A practical problem is that often an existing algorithm needs to be adapted to a slightly different setting. The advantage of our approach is that the designer can (sometimes) adapt the *building blocks* of the algorithm, and not the algorithm itself, letting the tool find algorithms with the adapted building blocks. We consider two adaptations of set algorithms: (i) priority queue, and (ii) a stack. In our variation of a priority queue, `add` corresponds to an `insert` in a priority queue, and adds elements with a unique priority key. The `remove` corresponds to a `deleteMin` operation, and removes the element with the lowest key (highest priority). This corresponds to changing the blocks of remove to only allow access to the first item in the list. After adapting the blocks and adjusting the sequential specification, we run the system and obtain 4 variations of a priority queue. Similarly, we repeat the process for stacks and obtain 8 variations, including the well-known Treiber algorithm [27]. For space reasons, we do not show the results in the paper.

4.5 Experimental Results

Table 1 is a summary of the major exploration experiments performed in the derivation process. Every row in the table corresponds to an exploration starting from a single schema. We report how many algorithms passed the correctness checks (Accepted Instances), the total time of exploration in minutes, the total number of instances that could have been checked under a naive exploration procedure (Total Instances), and what percentage of the space was actually checked by the the system (number of algorithms is shown in parentheses). Note that in each exploration experiment, we are using the method for automatically checking linearizability (see Section 5.1). This is because the tool explores many instances for each schema and the linearization points for each instance can be different. In cases where we found an instance (algorithm) that we thought was interesting (such as the one in Fig. 11), we checked that instance by specifying its linearization points (see Section 5.2). The effectiveness of the domain specific exploration procedure is apparent from the percentage of algorithms being checked. In all cases, it is less than 3% of the total number of algorithms. Our exploration procedure can be parallelized by exploring different assignments separately. This might reduce the opportunities for reduction of the space explored, but is expected to yield an overall improvement in the running times.

4.6 Summary

In this section we started with a sequential algorithm. Using an optimistic concurrency transformation we arrived at an intermediate schema. The challenge there was to find correct restarting validation conditions with respect to linearizability. Starting with the

[1] as suggested by [21], the marked bit can be stored in a low order bit of the next pointer as pointers are at least word aligned on most systems.

Schema	Accepted Instances	Time (Min.)	Total Instances	Checked Instances %
Null-based	16	26	64,512	0.088 (57)
Marked-based	6	15	138,240	0.048 (66))
Prio-Q	4	2	4,608	0.67 (31)
Stack	8	2	2,560	2.27 (58)

Table 1. Exploration Results.

weakest possible validation conditions (e.g. all set to *true*), we used the tool to produce a counter-example. By manually observing the counter-examples we found some of the simpler conditions. Unfortunately, these conditions were not enough to guarantee correctness and we had to strengthen them further. Towards that end, we proposed two insights (using $null$ or $marked$ for removal) and specified the building blocks corresponding to each. We than ran our tool which found a number of linearizable algorithms with various degrees of atomicity for each insight. We then took the best results of each exploration and manually applied transformations in order to obtain CAS and DCAS versions. It seemed that this step can be automated. Finally, we adapted some of the set algorithms to simpler data structures such as priority queues and stacks. To what extent these steps can be automated remains a fascinating item of future work.

5. Checking Linearizability

As mentioned in Section 2, linearizability is verified 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.

Checking linearizability boils down to finding a linearization for every concurrent execution. Towards this end, we employ a model checker as follows. The model consists of three parts: the concurrent object (e.g. a set algorithm), the executable sequential specification of the concurrent object and the *most general client* of that concurrent object [2]. The client operates by non-deterministically selecting the operations and the key values, thus exploring all possible sequences of operations. This client is executed by each thread. We check that every concurrent execution explored by the model checker is linearizable.

The checking is supported via two complementary approaches. The first method is an entirely automated one, requires no user annotations and is discussed in Section 5.1. This method is the key for the success of the systematic derivation process, as it requires no algorithm-specific user annotations. The second method discussed in Section 5.2 requires algorithm-specific user annotations, but allows the model checker to explore a larger state space of the algorithm than the first.

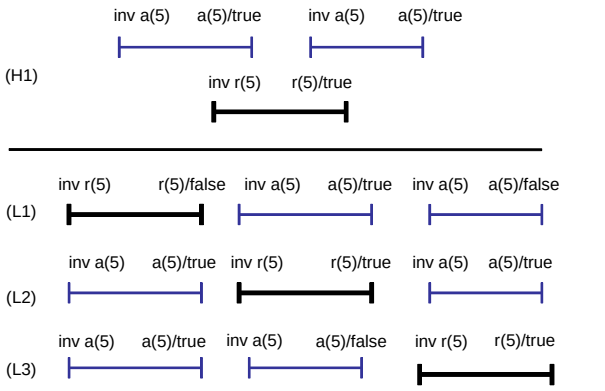

Figure 12. Enumeration of linearizations. (H1) is a concurrent history, and (L1),(L2), and (L3) are its three potential linearizations.

5.1 Checking Linearizability by Recording Histories

In this approach, for every history h, the system explores *all* possible linearizations, trying to find one that satisfies the sequential specification. The worst case time of this approach is exponential in the length of the history, as it may have to try all possible permutations of h.

Technically, the concurrent history h is obtained by instrumenting the schema to record h as part of the state. Upon invocation of an operation, we record the invocation event together with its arguments. Upon operation completion, we record the response event together with the result. Note that this is done only once per schema and is shared between all instances of that schema. However, recording the concurrent history as part of the state leads to an unbounded number of states that results in non-termination of the model checker. In order to bound the state space and guarantee termination, the most general client is parameterized by the maximum number of operations each thread can invoke. In our experiments this bound is typically set to two or three.

Enumerating Linearizations When the model checker finishes the exploration of a concurrent execution, the recorded history h is part of the final state. At that point, a procedure external to the model checker, is invoked to check linearizability of h. The procedure takes as input the history h and a sequential specification of the concurrent object. The procedure then attempts to find a linearization of h. Note that there could be many possible linearizations of h, but it is enough to find a single witness. If a witness is found, the model checker continues checking the algorithm. Otherwise, the algorithm is rejected. The procedure also supports checking of other correctness criteria (e.g., operation-level serializability).

EXAMPLE 5.1. *Fig. 12 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 possible linearizations of H1. In this case, these are (L1), (L2) and (L3). These histories are obtained by re-ordering only overlapping operations of H1. For each of these three potential linearizations, we execute the operations over the (executable) specification, 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.*

5.2 Checking via Linearization Points

Recall that even though there could be many possible linearizations of a concurrent history h, finding a *single* linearization is enough to declare the history correct. While requiring no user annotations, the main shortcoming of the previous approach is that it records the entire history as part of the state. 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. To perform checking, every time $lp(op)$ is reached in h, the operation *op* in the sequential specification is executed and the results are compared. If the results are the same, the exploration process continues, otherwise an error is raised.

EXAMPLE 5.2. *Consider the algorithm of Fig. 9, and an* `add` *operation. When the* `add` *succeeds (and returns* true*), the linearization point is at the statement* $pred{->}next = entry$*. This is the point at which the update of shared information takes place. When* `add` *returns* false*, the linearization point is at the last execution of the statement* $curr = curr{->}next$ *in* LOCATE*. This is the point where the negative outcome of the operation is already determined.*

This approach typically requires an insight on how the algorithm operates. Its advantage however is that because the linearization is built and checked on-the-fly, we no longer need to record h as part of the state. In turn, this allows to check algorithms with each thread executing the most general client, without a bound on the number of operations. The most general client is a program that non-deterministically selects the operations and key values that are used with the concurrent object.

Multiple Points Note that there could be more than one linearization of a concurrent history h. Therefore, there could be more than one linearization point $lp(op)$ for each operation *op*. For simpler algorithms, $lp(op)$ is usually a point in the code of *op*. But for more complex fine-grained algorithms, there may be several linearization points for $lp(op)$ which may reside in method(s) other than *op*. The choice of which point is selected is conditional on the history h.

EXAMPLE 5.3. *Consider the algorithm in Fig. 11. The* $lp(remove)$ *when* `remove` *returns* true *resides in the code of* `remove` *and is the physical removal of the node, that is, building block* $R6$*. Now consider the case where* `remove` *returns* false *for key* k*. The* $lp(remove)$ *in this case is conditional on the order and types of operations executed by other threads during the execution of* `remove`*. It is the earliest of the two points: (i) right* before *a successful addition of* k *by another thread and (ii) execution of building block* $R2$ *(comparing the key).*

Typically, in order to check algorithms where the linearization points occur in another thread, additional instrumentation of the model is required. We have performed this instrumentation for the algorithm in Fig. 11.

6. Related Work

There is a significant body of work describing concurrent objects as well as various formal methods for checking their correctness. Due to space reasons, we only survey some of this work.

Concurrent Set Algorithms The algorithm of [12] was the first lock-free set algorithm using only CAS instructions. This algorithm relies on a garbage collector (GC). Later, [21] introduced another lock-free set algorithm using explicit memory management. The introduction of manual memory management introduces many complexities that are beyond the scope of this paper. In [13], the authors present a concurrent algorithm with a wait-free `contains` also assuming a GC. The `add` and `remove` operations are blocking, due to the use of locks. The best algorithm derived in this paper uses only CAS operations, assumes GC, and also supports a wait-free `contains`. Similarly to [13], it is blocking in both `add` and `remove`. The reason is because once a node is marked, the thread who marked the node could crash before it physically removes the node. The marked node could then stay in the list permanently which may cause other operations to restart indefinitely. This scenario technically precludes the algorithm from being referred to as lock-free. This case can be avoided by requiring each operation to physically remove a marked node once it encounters it during `locate`. This will result in a different linearization point for a successful remove [18].

Exploration and Derivation In previous work [29], we also used a semi-automated approach for exploring a space of concurrent GC algorithms. That work used a limited search procedure and an abstraction specifically geared towards the safety property required for that specific domain. In this work, we concentrate on exploring and checking a broader class of concurrent algorithms. To that end, we define a more general exploration mechanism and provide an automatic procedure for checking linearizability, a property relevant to a wide class of concurrent objects.

There has been a considerable amount of work on using brute-force search to find a correct completion of a *partial program* (e.g., Sketching [25, 26, 24]), a "superoptimized" code sequence [3, 19], or an algorithm within a limited family (e.g., mutual exclusion algorithms [4]). Our approach differs in several ways, notably:

- *Systematic Derivation* We follow a systematic derivation that combines manual steps with automated exploration steps performed by PARAGLIDER. This clarifies which decisions are fundamental to the algorithm and which are a result of a specific implementation (e.g. merging the pointer with a marked bit).
- *Checking Linearizability* Automatic checking of properties relevant to concurrent algorithms such as linearizability.
- *Domain-specific Exploration* The ability to explore and check a large space of algorithms is due to the domain-specific exploration that leverages relationships between algorithms.

[1] introduces a formal derivation of concurrent queue algorithms close to the ones of [20]. Their derivation is an attempt to reconstruct known algorithms. In contrast, we explore a wide range of algorithms, and use an automated tool for exploring alternatives.

Checking Linearizability and Related Conditions CheckFence [6] checks the correctness of algorithms using symbolic bounded test-cases. It first constructs all sequential executions for operations invoked in the test program and creates a set of correct observable behaviors (sequences of invokes and returns). It then checks that every concurrent executions is observationally equivalent to some correct (sequential) behavior. In [6], algorithms were tested using symbolic test-cases. Previous work by the same authors [5] checked for operation-level sequential consistency using bounded test-cases and user provided specification of "commit points". In contrast, we assume a sequentially consistent memory model, and check for *linearizability*. Moreover, when the user specifies fixed linearization points, our approach uses test-case invoking an *unbounded sequences of operations* (over an underlying data-structure of a bounded size).

Verification Of Linearizability There have been several works on formally verifying linearizability. The algorithm of [13] has been verified by [7] using an automata based approach and by [28] using the rely-guarantee proof method. In the work of [2], the authors assumed a *fixed number of threads* and using abstract interpretation verified some small concurrent algorithms. These proof methods typically require manual intervention, such as specifying linearization points and/or constructing the actual proof. As noted by [7], this is especially difficult when the linearization point for a method does not reside in its code and is dependent on the program trace. This is the case for several of the algorithms derived in this paper as well as the `contains` of [13]. We see our approach as complementary to a formal verification of a final algorithm. The system produces fine-grained candidates which could then be further verified formally if necessary. The work of [10] provides bounded model checking of commit-atomicity for small algorithms, assuming user-provided *fixed commit points*, which reside in the code of the method. Our method for checking fixed points does not require the point to reside in the method and hence we can check more complex algorithms. Wing and Gong [30] provide a simulation procedure for testing linearizability with user provided or randomly generated test-cases. A concrete test case is executed in a simulator and the system attempts to find a linearization of one particular execution. The key advantage of our system is that up to a bound, all possible test cases are automatically generated, and for each test case, exhaustively, all possible executions are checked.

7. Conclusion and Future Work

We presented a systematic construction of several linearizable concurrent algorithms. Assisted by a new tool, we were able to arrive at practical concurrent algorithms while clearly distilling the algorithmic insights from implementation details. In particular we derived a practical set algorithm that only uses CAS, and provides a wait-free `contains` operation.

While modest, this step is encouraging in further exploring algorithmic construction mechanisms. In the future, we plan to formalize the informal derivation steps presented in this paper and extend the construction mechanism to arrive at verified rather than checked algorithms.

Acknowledgements

We thank Maged Michael, Dragan Bosnacki, Gerard Holzmann, Mooly Sagiv, Greta Yorsh, David Bacon and Noam Rinetzky.

References

[1] ABRIAL, J.-R., AND CANSELL, D. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS 11*, 5 (2005), 744–770.

[2] AMIT, D., RINETZKY, N., REPS, T. W., SAGIV, M., AND YAHAV, E. Comparison under abstraction for verifying linearizability. In *CAV* (2007), vol. 4590 of *LNCS*, Springer, pp. 477–490.

[3] BANSAL, S., AND AIKEN, A. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev. 40*, 5 (2006), 394–403.

[4] BAR-DAVID, Y., AND TAUBENFELD, G. Automatic discovery of mutual exclusion algorithms. In *Proc. of the symp. on Principles of Distributed Computing* (2003), pp. 305–305.

[5] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV* (2006).

[6] BURCKHARDT, S., ALUR, R., AND MARTIN, M. M. K. Checkfence: checking consistency of concurrent data types on relaxed memory models. *SIGPLAN Not. 42*, 6 (2007), 12–21.

[7] COLVIN, R., GROVES, L., LUCHANGCO, V., AND MOIR, M. Formal verification of a lazy concurrent list-based set algorithm. In *CAV* (2006).

[8] DOHERTY, S., DETLEFS, D. L., GROVES, L., FLOOD, C. H., LUCHANGCO, V., MARTIN, P. A., MOIR, M., SHAVIT, N., AND GUY L. STEELE, J. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA* (2004), pp. 216–224.

[9] ELMAS, T., TASIRAN, S., AND QADEER, S. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI* (2005), pp. 27–37.

[10] FLANAGAN, C. Verifying commit-atomicity using model-checking. In *SPIN* (2004).

[11] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. *SIGPLAN Not. 38*, 11 (2003), 388–402.

[12] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proc. of conf. on Distributed Computing* (London, UK, 2001), Springer, pp. 300–314.

[13] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *Proc. of conf. On Principles Of Distributed Systems (OPODIS 2005)* (2005), pp. 3–16.

[14] HERLIHY, M. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. 15*, 5 (1993), 745–770.

[15] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *Trans. on Prog. Lang. and Syst. 12*, 3 (1990).

[16] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.

[17] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programn. *IEEE Trans. Comput. 28*, 9 (1979), 690–691.

[18] MICHAEL, M. Personal communication.

[19] MASSALIN, H. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Proc. of conf. on Architectual support for programming languages and operating systems* (1987), IEEE, pp. 122–126.

[20] MICHAEL, M., AND SCOTT, M. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC* (1996).

[21] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *SPAA* (2002), pp. 73–82.

[22] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653.

[23] PRAKASH, S., LEE, Y. H., AND JOHNSON, T. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput. 43*, 5 (1994), 548–559.

[24] SOLAR-LEZAMA, A., ARNOLD, G., TANCAU, L., BODÍK, R., SARASWAT, V. A., AND SESHIA, S. A. Sketching stencils. In *PLDI* (2007), pp. 167–178.

[25] SOLAR-LEZAMA, A., RABBAH, R. M., BODÍK, R., AND EBCIOGLU, K. Programming by sketching for bit-streaming programs. In *PLDI* (2005), ACM, pp. 281–294.

[26] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *ASPLOS* (2006), pp. 404–415.

[27] TREIBER, R. K. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center, APR 1986.

[28] VAFEIADIS, V., HERLIHY, M., HOARE, T., AND SHAPIRO, M. Proving correctness of highly-concurrent linearisable objects. In *PPoPP* (2006).

[29] VECHEV, M. T., YAHAV, E., BACON, D. F., AND RINETZKY, N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* (2007), pp. 456–467.

[30] WING, J. M., AND GONG, C. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput. 17*, 1-2 (1993), 164–182.