Proving Concurrent Data Structures Linearizable

Vineet Singh
University of California, Riverside
Riverside, CA, USA
Email:vsing004@cs.ucr.edu

Iulian Neamtiu
New Jersey Institute of Technology
Newark, NJ, USA
Email: ineamtiu@njit.edu

Rajiv Gupta
University of California, Riverside
Riverside, CA, USA
Email: gupta@cs.ucr.edu

***Abstract*—Linearizability of concurrent data structure implementations is notoriously hard to prove. Consequently, current verification techniques can only prove linearizability for certain classes of data structures. We introduce a *generic*, *sound*, and *practical* technique to statically check the linearizability of concurrent data structure implementations. Our technique involves specifying the concurrent operations as a list of sub-operations and passing this specification on to an automated checker that verifies linearizability using relationships between individual sub-operations. We have proven the soundness of our technique. Our approach is expressive: we have successfully verified the linearizability of 12 popular concurrent data structure implementations including algorithms that are considered to be challenging to prove linearizable such as *elimination back-off stack*, *lazy linked list*, and *time-stamped stack*. Our checker is effective, as it can verify the specifications in less than a second.**

I. Introduction

Linearizability, introduced by Herlihy and Wing [1], is the standard form of correctness for concurrent data structure implementations. Linearizability means that the entire observable effect of each operation on a concurrent data structure happens instantly, i.e., the effect of each operation is atomic. Concurrent implementations of abstract data structures (stacks, queues, sets, etc.) are becoming more and more complex as implementations that increase the degree of concurrency are identified. This in turn is making linearizability verification harder. Even recent, state-of-the-art techniques (e.g., [2], [3]) lack generality as they are limited to specific classes of concurrent data structures — so far no technique (manual or automatic) for proving linearizability has been proposed that is both sound and generic. To this end, we present a generic technique for proving linearizability that is independent of any properties of the concurrent data structure.

Prior work on atomicity verification [4], [5] has utilized the concept of *moverness* [6]. Moverness has also been used to verify linearizability of concurrent operations [7]. The main idea is to prove that individual atomic actions in a concurrent operation can commute with atomic actions from other threads. The operation is transformed through a series of steps to reach an equivalent single atomic action. The moverness requirement of an action is global with respect to all other atomic actions present in the program. This makes moverness too strong a requirement for complex concurrent operations.

Instead, we present a verification technique which can be applied to a wide range of concurrent data structure implementations. A concurrent execution of operations can be modeled as interleaved sequences of corresponding atomic sub-operations. The novelty of our approach is that:

1. We can express the set of all sequences allowed by an implementation in terms of relationships between pairs of individual sub-operations.

2. Given the properties of sub-operation pairs, we can statically verify if all the sequences of sub-operations allowed by the implementation can be mapped to an equivalent non-interleaved sequence of sub-operations while maintaining the order of non-overlapping operations, i.e., linearizability.

Our technique consists of (1) a specification language that allows concurrent data operations to be specified simply as sequences of atomic sub-operations and (2) a static checker that, given the relationship between the sub-operations, determines if the implementation is linearizable. If the linearizability proof fails, the static checker returns a sequence of sub-operations that could not be linearized.

We have applied our technique to 13 popular concurrent data structure implementations and were able to verify 12 of them (as our approach is sound, the one false positive is explained in Section VIII). The evaluation shows that our technique is generic, practical, and efficient. Our contributions include:

- **A model to express execution history** of concurrent operations in terms of static properties of atomic sub-operations (with no limit on the number of concurrently executing operations).
- **A static linearizability checking technique** that, given a concurrent data structure specification in terms of sub-operations and relationship of in-between sub-operations, automatically checks linearizability.
- **A concurrent data structure specification language** that allows the user to easily and concisely express concurrent operations as sequences of atomic sub-operations.
- A **proof of soundness** for our technique.
- An evaluation on 13 well known concurrent data structure implementations – 12 were verified to be linearizable.

II. System Model and Linearizability

A concurrent data structure *implementation* consists of a shared state (defined by shared variables) and methods which operate on the shared state. An *execution* consists of a variable number of threads, each executing one of the defined methods. An *operation* is a successful execution of a method. The concept of operations is native to linearizability of concurrent implementations. Most of the prior work on concurrent data structures with fixed linearization points establish the linearizability of the data structure by locating the linearization point of the operations present in the implementation. For example, Michael and Scott [8] have proven their queue's linearizability by locating the linearization points of enqueue, dequeue_empty, and dequeue_non_empty operations. Hendler et al. [9] have also used the operations' linearization points for proving linearizability. Operations are sequential compositions of atomic sub-operations. Each sub-operation (atomic) α has the form $< g > t$, where g is a pre-condition to the sub-operation. A sub-operation can execute only at a state which satisfies g, while t is the set of reads and writes which get executed when α gets executed. Sub-operations can be *global* or *local*. A global sub-operation involves reading or writing shared variables (or references). Local sub-operations are used just for the sake of completeness and do not play any role in proving linearizability. Section IV gives a detailed language along with examples to help user express concurrent data structures implementations as a sequence of sub-operations.

A. Execution Model

We assume a sequentially consistent memory model. The program state s is the current valuation of the variables (both shared and local) present. The program state changes with execution of sub-operations belonging to operation instances. Each operation instance is a unique invocation of one of the operations defined in the specification. Each operation instance has a unique operation instance id from the set O_{id}. We represent sub-operation $\alpha(< g > t)$ being executed as a part of operation instance v as $\alpha[v]$. $g[v]$ and $t[v]$ represent the corresponding pre-condition and sub-operation body. Note that we do not use *thread id* instead of operation instance id because more complicated data structures ([8], [9]) can have an operation execution distributed among multiple threads. The atomicity and error conditions are given in Figure 1. We use the notation $(s_1, H_1, \alpha) \rightarrow (s_2, H_2)$ to indicate that sub-operation α executed at program state s_1 takes the program state to s_2 while the execution history changes from H_1 to H_2. The definition ATOMIC states that if the current program state s_1 satisfies the sub-operation pre-condition $g[v]$, $(v \in O_{id})$, represented by $s_1 \models g[v]$, the program state is modified with transition $t[v]$. The execution history H_1 gets appended with sub-operation $\alpha[v]$. The ERROR definition states that the sub-operation α cannot successfully execute at a program state where the corresponding pre-condition is not satisfied.

$$\boxed{(s_1, H_1, \alpha) \rightarrow (s_2, H_2)}$$

$$\texttt{ATOMIC}\ \frac{s_1 \models g[v] \qquad (s_1, t[v]) \rightarrow s_2}{(s_1, H, \alpha[v]) \rightarrow (s_2, H.\alpha[v])}$$

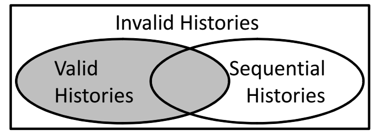
$$\texttt{ERROR}\ \frac{s_1 \models \neg g[v]}{(s_1, H, \alpha[v]) \rightarrow (\text{error}, H)}$$

Figure 1. Atomicity and error definitions.

B. Histories

Definition 1. A *history* H is a finite sequence of sub-operations where the sub-operations corresponding to the same operation instance follow the program order.

A history H is *sequential* if the sub-operations from the same operation instance occur together. A sequential history is *legal* if it follows the abstract data structure behavior. A history H in which the preconditions for all the sub-operation instances present in the history are satisfied is *valid* with respect to the implementation (naturally, we call "invalid" a history that is not valid). The following Venn diagram shows the relationship between valid, invalid and sequential histories.



Two valid execution histories are *equivalent* if they contain the same sub-operation instances and the program state visible to a sub-operation instance is the same in both the histories.

C. Linearizability

Linearizability requires every execution history to be equivalent to some legal sequential history that preserves the order of non-overlapping operations in the original history.

Many techniques are available for checking the correctness of a sequential implementation with respect to the abstract data structure (e.g., [10]). Taking advantage of this we make the following safe assumption in our technique:

Assumption 1. Every valid sequential history with respect to the implementation is a legal sequential history.

In other words, we assume that all sequential executions of the implementation are correct. With this assumption, our linearizability definition is:

Definition 2. An implementation is *linearizable* if for any valid history there exists some equivalent valid sequential history such that the order of non-overlapping operations in the two histories is the same.

III. Overview and Example

We illustrate our technique on the MS non-blocking queue [8]: a singly linked list-based queue with three operations, initialize, enqueue, and dequeue (Figure 2). Each queue node has next and value fields. Enqueue first allocates a new node then reads the tail pointer and sets the next pointer of the end node to point to the newly allocated node; the final step in the process is to set the tail pointer to the new node. Any thread operating on the list can set the trailing tail pointer. Dequeue reads head and tail pointers then checks if both point to the same node. If head and tail point to the same node, i.e., the queue is empty, dequeue returns false; otherwise, dequeue reads the head node's value and updates the head pointer.

a) Specifying concurrent operations: The user specifies the concurrent data structure's operations as a sequence of atomic sub-operations. Let us consider the *enqueue* method; the `loop` in the method leads to an unbounded number of execution paths. We leverage the notion of pure loops ([11], [4]) to transform the loop to its last iteration. Figure 3 column 1 shows one of the possible loop free execution paths of the *enqueue* method. Figure 3 column 2 shows CAS (Compare and Swap) replaced with corresponding sub-operation $< g > t$ format. The $next$ var in line 5 and 6 is a local variable

```
typedef struct Node_t * Node
struct Node_t{
    int value;
    Node next;
    }
struct Queue{
    Node Head, Tail;
    } * Q;

initialize(Q: pointer to queue)
    node = new_node()
    node→next = NULL
    Q→Head = Q→Tail = node
```

```
enqueue(Q:pointer to queue, value:int)
    node = new_node();
    node→value = value;
    node→next = NULL;
    loop
    tail = Q→Tail;
    next = tail→next;
    if tail == Q→Tail then
        if next == NULL then
            if CAS(&tail→next, next, node) then
                break;
            end if
        else
            CAS(&Q→Tail, tail, next);
        end if
    end if
    endloop
    CAS(&Q→Tail, tail, node);
```

```
dequeue(Q:pointer to queue, pvalue:int): boolean
    loop
    head = Q→Head;
    tail = Q→Tail;
    next = head→next;
    if head == Q→Head then
        if head == tail then
            if next == NULL then
                return FALSE;
            end if
            CAS(&Q→Tail, tail, next);
        else
            pvalue = next→value;
            if CAS(&Q→Head, head, next) then
                break;
            end if
        end if
    end if
    endloop
    free(head)
    return TRUE;
```

Figure 2. Michael and Scott non-blocking concurrent queue [8].

```
1. node X = new_node();
2. node tail = Q→tail;
3. node next = tail→next;
4. assume (tail == Q→tail);
5. assume (next == NULL);
6. CAS(tail→next, next, X);
7. CAS(Q→tail, tail, X);
```

```
2. node tail = Q→tail;
3. node next = tail→next;
4. assume (tail == Q→tail);
5. assume (next == NULL);
6. <tail→next == next>
   tail→next = X;
7. <Q→tail == tail>
   Q→tail = X;
```

```
node tail = Q→tail;
assume (tail == Q→tail);
<tail→next == NULL>
    tail→next = X;
<Q→tail == tail>
    Q→tail = X;
```

```
<Q→tail = tail; tail→next == NULL>
    tail→next = X;
<Q→tail == tail>
    Q→tail = X;
```

Figure 3. Expressing an operation as a sequence of atomic sub-operations.

```
T1.    struct node{int value, node next};
T2.    struct queue{node head, node tail};
G1.    queue Q ;
Enqueue (int val)
E1.    node X;
E2.    node Z;
E3.    1 X = malloc node;                                   ...a
E4.    1 X.value = val;                                     ...b
E5.    1 X.next = NULL;                                     ...c
E6.    1 < Q.tail == Z;  Z.next == NULL;>
                 Z.next = X;                                ...d
E7.    * < Q.tail == Z>
                 Q.tail = X;                                ...e
Dequeue_nonempty()
D1.    node X;
D2.    1 < Q.head == X;  Q.tail ≠ X;   X.next≠ NULL;>
                 Q.head = X.next;                           ...f
D3.    1 free X;                                            ...g
Dequeue_empty()
DE1.   node X;
DE3.   1 < Q.head == X;   Q.tail == X;   X.next == NULL;>
```

Figure 4. MS non-blocking queue [8] specification.

i.e., cannot be modified by other threads. Replacing the value of $next$ with NULL gives us column 3. The only possible modification to Q→tail in the program is setting the *next* node in the list. The value of Q→tail does not change between 4 and 6 because that would violate the pre-condition tail→next == NULL. Adding this pre-condition to the sub-operation gives us column 4. Previous works on atomicity verification ([11], [5], [4], [12], [13]) have detailed discussion about program transformations required to express operations as sequences of atomic sub-operations. We are skipping this discussion here for brevity.

Figure 4 contains the MS queue specification in our language. The user specifies data structures by declaring their type name along with the fields. For example, line T1 declares a type *node* which has two fields and line G1 declares a shared variable **Q** of type *queue*. There are two different *dequeue* specifications, *Dequeue_nonempty* and *Dequeue_empty*, which correspond to the sequence of sub-operations for dequeue on a non-empty queue and an empty queue, respectively. Figure 5 shows the global sub-operations that involve shared variables, i.e., the global sub-operations of MS queue.

b) Pairwise ordering and reversibility: An ordered pair of sub-operations cannot be part of a valid history if the execution of the first sub-operation destroys the precondition for the second one. For example, sub-operation d in Figure 4 line E6 cannot be followed by another instance of d because the first instance sets the tail→next pointer to a non-null value, invalidating the precondition for the second instance. Figure 5 shows all the pairs of sub-operations and their feasibility. In our approach, $a \wr b$ means sub-operation a does not destroy the precondition for b. The sub-operation execution order in a feasible (pair-wise orderable) pair can be changed if changing the order will not affect the value of shared or local variables. We call this *reversibility*, denoted as $a \ominus b$. The properties are explained in detail in Section V.

A boundary case refers to a pair of sub-operations that includes the last sub-operation of an operation followed by the first sub-operation of an operation. Pair-wise orderable non-boundary pairs lead to interleaving operations. Boundary pairs are always defined as non-reversible. Pairwise reversibility for non-boundary sub-operation pairs for MS queue is also defined in Figure 5.

c) Trace transformation: An execution history formed by moving sub-operations using the reversibility property is equivalent to the original history. A valid concurrent history can be mapped to a valid sequential history using this trace transformation. The order of non-overlapping operations always remains the same during this process because the

Global sub-ops for Enqueue and Dequeue

$Enqueue()$
1 < Q.tail == Z; Z.next == NULL;>
  Z.next = X; ...d
* < Q.tail == Z;>
  Q.tail = X; ...e
	$Dequeue\_nonempty()$
	1 < Q.head == X; Q.tail $\neq$ X; X.next$\neq$ NULL;>
	  Q.head = X.next ...f
	
	$Dequeue\_empty()$
	1 < Q.head == X; Q.tail == X; X.next == NULL;> ...h

	Pair-wise ordering		Non-Boundary Cases	Pair-wise reversibility
$\neg d \wr d$	$d \wr f$ [if($Q.head \neq Q.tail$)]	$f \wr f$ [Boundary case]	d, f	$d \ominus f$
$\neg d \wr e$	$e \wr d$ [Boundary case]	$f \wr h$ [Boundary case]	f, e	$f \ominus e$
$\neg d \wr h$	$e \wr f$ [Boundary case]	$h \wr d$ [Boundary case]		
$\neg e \wr e$	$f \wr d$ [Boundary case]	$h \wr h$ [Boundary case]		
$\neg e \wr h$	$f \wr e$			
$\neg h \wr e$	$\neg h \wr f$			

Figure 5. Proving the MS queue linearizable.

boundary pairs are defined to be non-reversible. For example, consider a valid history for the MS queue:

$$..., d_1, f_2, f_3, e_1, ...$$
$$\equiv ..., d_1, f_2, e_1, f_3 ... \text{ (using } f_3, e_1 \equiv e_1, f_3 \text{)}$$
$$\equiv ..., d_1, e_1, f_2, f_3 ... \text{ (using } f_2, e_1 \equiv e_1, f_2 \text{)}$$

For a linearizable implementation, every valid concurrent history can be mapped to an equivalent valid sequential history using trace transformation. Our linearizability checker in Algorithm 1 answers the question: "Given an ordering and reversiblity specification, can all the valid histories (for unbounded number of concurrent operation instances) be mapped to an equivalent sequential history using trace transformation?"

IV. Specification Language

We have designed a language to assist the user in expressing concurrent data structure operations in terms of sub-operations. Note that the language is not central to our technique — our technique will work as long as concurrent operations can be expressed as sub-operations with ordering and reversibility properties. That said, we found the language made the task of expressing sub-operations and finding the pair-wise ordering and reversibility very easy. In this section we first define the language and then demonstrate its use for expressing common features in concurrent implementations.

A. Syntax

We provide the user with a simple C-like syntax, shown in Figure 6. Specifications consist of an optional list of structure declarations st, global declarations g and a list of operations op. Each global declaration consists of a variable name var and its type. Types can be **int** or **struct**, where **struct**s have a name $sname$ and consist of a list of fields; each fields has a name $fname$ and a $type$.

Operations: Each operation op has a name $oname$, and an optional argument ($type\ var$). Operation bodies $opBody$ consist of local variables, $dList$, and sub-operations, $sList$.

Sub-operation: A sub-operation specification has a **tmark** followed by a pre-condition and a sub-operation body. $tmark$ marks the thread which executes the statement. A $tmark$ of **1** means that the thread invoking the operation instance will perform the sub-operation. If a $tmark$ is *****, it indicates that any thread can perform the sub-operation. Specifying $tmark$ for each sub-operation allows specifying an operation which is distributed across multiple threads.

Specification	sp	::=	$stList\ g\ opList \mid g\ opList$
Struct Decl	$stList$	::=	$stList\ st \mid st$
	st	::=	struct $sname$ $\{l\}$;
	l	::=	$l, type\ fname \mid type\ fname$
Type	$type$	::=	int $\mid sname$
Global Decl	g	::=	$dList$
Operation	$opList$	::=	$opList\ op \mid op$
	op	::=	$opName\ opBody$
	$opName$	::=	$oname(dList\ ) \mid oname$
	$opBody$	::=	$dList\ sList$
	$sList$	::=	$sList\ subOp \mid subOp$
	$dList$	::=	$dList\ decl \mid decl$
Declaration	$decl$	::=	$type\ var$; | $type\ var$[ ] ;
	$subOp$	::=	$tmark\ subOpBody$
		|	$tmark\ < cList >\ subOpBody$
		|	$tmark\ < cList >$
	$tmark$	::=	1 | $*$
Pre-condition	$cList$	::=	$cList\ condition$; | $condition$;
	$condition$	::=	$lhs\ rop\ rhs$
	$subOpBody$	::=	$subOpBody\ stmnt$; | $stmnt$;
	$stmnt$	::=	var = malloc $type$ | free var
		|	$lhs\ =\ rhs$
		|	READ lhs
LHS	lhs	::=	$var \mid var\ .\ fname \mid var[index]$
RHS	rhs	::=	ID | n | NULL | lhs
	$index$	::=	$n \mid var$
Operators	rop	::=	$== \mid \neq \mid \leq \mid \geq \mid < \mid >$
Integers	n		
Variable	var		
Struct Name	$sname$		
Field Name	$fname$		
Op Name	$oname$		

Figure 6. Syntax of Specification Language.

– An optional precondition to the sub-operation is composed of a list of $condition$s, where each condition is a relational expression involving a variable **var**, or field **var.fname** on lhs and null, constant, variable, or field on rhs.

– A statement can be allocation or deallocation; malloc and free statements are used for specifying local sub-operations. Other possible statement forms involve assigning values to a variable or writing a field. **READ** refers to reading of a field or a variable.

– We use **ID** to identify the operation instance; ID is useful in modeling locks, as explained shortly.

B. Modeling Synchronization Primitives

We now show how to model compare-and-swap (CAS), fetch-and-increment (F&I), and locks, using our language.

Modeling Compare-and-swap: CAS can be modeled in two different ways, depending on the implementation.

Case 1 - When the memory location being compared was read or written before CAS statement and execution of the CAS statement is conditionally controlled by an *if statement*. In this case the CAS statement can be modeled as a sub-operation with the *if condition* as the precondition and the memory write as the body of the sub-operation. For example, in Figure 2, the execution of the CAS statement on line 10 is dependent on the *if conditions* in lines 8 and 9. The specification for the CAS statement is as follows:

$$1 < \text{Q.tail == Z;} \quad \text{Z.next == NULL;}>$$
$$\text{Z.next = X;}$$

Case 2 - When the memory location being compared was read or written before CAS statement and execution of CAS statement is unconditional. In this case CAS statement can be modeled as a sub-operation with no precondition and two-statement body: the first statement reads/writes the location being compared while second statement writes the memory location. Figure 7 shows pseudocode excerpt from implementation of elimination back-off stack by Hendler et al. [9]. The second row shows corresponding sub-operation specification.

Pseudocode	him=collision[pos];
	while(!CAS(&collision[pos],him,mypid))
	him=collision[pos];

Specification	1 him = collision[pos]; collision[pos] = mypid;

Figure 7. Sample CAS specification from [9].

Modeling Fetch-and-Increment: F&I on a shared variable x is modeled by a sub-operation with no precondition. The body of the sub-operation consists of two statements, first reading shared variable x in a local variable and second incrementing x. Figure 8 left shows the pseudo code from the *enqueue* operation in Herlihy and Wing's queue [1] which uses fetch-and-increment. The right side shows the specification in our language — declaration of variables *back* and *AR* and the *enqueue* operation.

Pseudocode	Specification
enqueue(lv){ /*(Fetch and Increment)*/ (k, back) := (back, back +1); AR[k] := lv;}	int AR[ ]; int back; $Enqueue$(int lv) int k; 1 k = back; back = back + 1; 1 AR[k] = lv;

Figure 8. Sample Fetch-and-increment specification from [1].

Modeling Locks: A lock can be modeled using our specification language as a shared variable with a default value. The locking sub-operations will look like

$$1 < \text{lock == default\_value;} >$$
$$\text{lock = ID;}$$

where operation ID refers to a unique value identifying the operation instance. Any sub-operation performed with the lock acquired will have a precondition of the form:

$$< \text{lock == ID;}>$$

Unlocking takes the form:

$$1<\text{lock == ID;}>$$
$$\text{lock = default\_value;}$$

V. Proving Linearizability

The number of possible valid histories for an implementation is directly proportional to the number of executing concurrent operation instances. The number of valid histories becomes intractable with the increase in the number of concurrent operation instances executing on the data structure. We solve the problem of intractable number of histories by breaking the history down into basic building blocks, *sub-sequences of two sub-operations*. We then argue about all the histories in terms of properties on these building blocks. We call (a, b) a *sub-operation pair*, where a and b *belong to different operation instances*; for brevity, we will heretofore drop the parentheses when referring to pairs. Note that a and b can be the same sub-operation. The number of possible pairs for an implementation with n sub-operations is n^2.

We define the property of pair-wise ordering as follows:

Definition 3. Figure 9 shows how we determine if a sub-operation pair is orderable. The rule states that a sub-operation pair α_1, α_2 is *not pairwise orderable* (denoted by $\neg\alpha_1 \wr \alpha_2$) if after executing α_1, the pre-condition for α_2 is not met; otherwise $\alpha_1 \wr \alpha_2$.

$$\frac{\forall s, \forall u, v \in O_{id}, u \neq v \qquad (s, H, \alpha_1[u]) \rightarrow (s_2, H_2) \qquad (s_2, H_2, \alpha_2[v]) \rightarrow (\text{error}, H_2)}{\neg\alpha_1 \wr \alpha_2}$$

Figure 9. Pair-wise ordering.

For example, consider the sub-operation d of MS queue's *enqueue* (Figure 5):

$$1 < \text{Q.tail == Z;} \quad \text{Z.next == NULL;}>$$
$$\text{Z.next = X;}$$

The pre-condition states that Q→tail→next should be NULL. Now consider the pair d, d (d_1, d_2 for clarity). The execution of d_1 sets the value of Q→tail→next to a non-NULL value. This invalidates the pre-condition for sub-operation instance d_2. Hence d, d is not pair-wise orderable; i.e., $\neg d \wr d$.

Figure 10 states the rule for determining sub-operation pair's reversibility.

Definition 4. A sub-operation pair α_1, α_2 is *reversible* (denoted by $\alpha_1 \ominus \alpha_2$) iff
1. $\alpha_1 \wr \alpha_2$ and $\alpha_2 \wr \alpha_1$ and
2. Given a program state, the final program state is the same irrespective of the order of execution of α_1 and α_2.

Note that pair reversibility is different from *moverness* properties (left movers and right movers). The left and right mover properties of an atomic sub-operation are very restrictive as the sub-operation should be commutative with respect to every sub-operation present in the program. Reversibility on the

$$\frac{\forall s, \forall u, v \in O_{id}, u \neq v \qquad \dfrac{\alpha_1 \wr \alpha_2}{\begin{array}{c}(s, H, \alpha_1[u]) \rightarrow (s_2, H_2)\\ (s_2, H_2, \alpha_2[v]) \rightarrow (s_3, H_3)\end{array}} \qquad \dfrac{\alpha_2 \wr \alpha_1}{\begin{array}{c}(s, H, \alpha_2[v]) \rightarrow (s_2', H_2')\\ (s_2', H_2', \alpha_1[u]) \rightarrow (s_3', H_3')\end{array}} \qquad s_3 = s_3'}{\alpha_1 \ominus \alpha_2}$$

Figure 10. Pair-wise reversibility.

other hand is a property on a single sub-operation pair. [12] has discussed that *movers* fail to prove atomicity in the presence of the ABA problem [14]; reversibility, on the other hand, enables our technique to handle the ABA problem.

Pairs can also be *conditionally orderable*. For example, consider the pair d, f, (Figure 5) where sub-operation d is:

1< Q[tail] == Z; Z.next == NULL;>
Z.next = X;

and sub-operation f is:

*< Q.head == X; Q.tail $\neq$ X; X.next$\neq$ NULL>
Q.head = X.next;

$d \wr f$ only if Q.head$\neq$Q.tail.

We consider a pair orderable, if it is orderable for any possible values of the variables involved.

The reversibility of a *conditionally orderable pair* is defined under the same conditions for which it is orderable. For example, reversibility for pair d, f (mentioned above), will be calculated with the premise that Q.head$\neq$Q.tail. Reversibility of a pair is decided conservatively. If the reverse order of execution is not equivalent to the original pair under any possible condition (which is satisfied by the ordering condition), we deem the pair to be non-reversible. We found that the conservative definition of reversibility is not sufficient to handle complex interactions of operation instances. We explain in Section VI how to handle such complex cases.

In order to preserve the order of non-overlapping operation during the trace transformation, we have used the following simple technique. A pair formed by last sub-operation of any operation and the first sub-operation of any operation (a.k.a boundary pair) is always set to be non-reversible. This insures that the order of non-overlapping operations in a history will not change when moving around the sub-operations using reversibility property. Given the pairwise ordering and pairwise reversibility of all possible pairs of sub-operations, we express a valid history as follows:

Definition 5. A *valid history H* (sequence of sub-operations following program order) is defined as follows:
1. For any sub-sequence a, b of H, either a and b belong to same operation instances, or $a \wr b$ and
2. For any sub-sequence a, b of H, where a and b belong to different operation instances, if $a \ominus b$ then the history formed by reversing the order of a and b in H is also a valid history.

The equivalence of histories is defined in terms of pairwise reversibility:

Definition 6. Two valid histories H and H' are *equivalent*, denoted H $\equiv$ H', iff H' can be formed from H by reversing the pairs present in H using pairwise reversibility.

Using this definition of equivalence of history, we redefine our problem of checking linearizability as follows:

Definition 7. An implementation is *linearizable* iff all valid histories with respect to the implementation can be mapped to some sequential history by changing the order of pairwise reversible sub-operations.

Note that the order of non-overlapping operations will always be preserved because boundary pairs are not reversible.

Algorithm 1 Checking Linearizability

```
1: Input: S: Set of Operations
2: Sub(S):Set of all sub-operations
3: first(S): set of first sub-operations for each operation in S
4: I is the set of all possible prefix sequences for operations
5: next(x), x∈ I : next sub operation in the operation after x
6: x∈ I, y∈I∪Sub(S), x.y: sequence formed by concatenating y after x
7: x∈ I, y∈I∪Sub(S), x≀y iff x.y is a valid history
8: x∈ I, x is a proper prefix, y∈I∪Sub(S), x⊖y iff x.y ≡ y.x
9: x∈ I, x is a complete operation, y∈Sub(S), y∉first(S), x⊖y iff x.y ≡ y.x
10: x∈ I, x is a complete operation, y∈I, ¬x⊖y
11: x∈ I, x is a complete operation, y∈Sub(S), y∈first(S), ¬x⊖y
12: Check_Linearizability()
13:    ret = TRUE
14:    for all x∈ I, x is a proper prefix do
15:       ret = ret && Check(x)
16:    end for
17:    RETURN ret
18: Check(x)
19:    for all y ∈ Closure(x) do
20:       if y≀next(x) && ¬y⊖next(x) then
21:          RETURN FALSE
22:       end if
23:    end for
24:    RETURN TRUE
25: Closure(u)
26:    C = {φ}
27:    for all w∈ I; st u ≀ w && ¬u ⊖ w do
28:       C = C ∪ w
29:    end for
30:    for all z ∈ C do
31:       for all w∈ I; st z ≀ w && z ⊖ w do
32:          C = C ∪ w
33:       end for
34:    end for
35:    RETURN C
```

Algorithm 1 presents our linearizability checking approach. The input to the algorithm is the set of operations, corresponding sub-operation sequences, pair-wise ordering as well as reversibility for each possible pair. We start by initializing set **I** with all prefixes of operations (line 2). The prefix of an operation o is a partial sequence of sub-operations starting from the first sub-operation of o following program order. The prefix set for an operation represented by a, b, c (where a, b and c are the sub-operations) is $\{a, ab, abc\}$.

We define the ordering for a prefix pair (x, y — x and y $\in$ **I**) simply by checking if the concatenated sequence of x and y is a valid history with respect to the input ordering and reversibility specifications (line 5). Reversibility for a prefix pair (x, y — x and $y \in$ **I**) is defined by the equivalence of two histories formed by reversing the order of x and y (line

```
type E{
  int mark;
  int key;
  E next;
  }
E H, T;
ExE locate(int k)
  E p = H;
  E c = p.next;
  while(c.key < k)
  p = c;
  c = p.next;
  endwhile
  return p,c;
bool contains(int k)
  ExE p,c = locate(k);
  bool b = (c.key == k);
  return b;
```

```
bool remove(int k)
  bool restart = true, retval;
  while (restart)
  ExE p,c = locate(k);
  atomic{
  if p.next == c && !p.mark then
    restart = false;
    if c.key == k then
      c.mark = true;
      p.next = c.next;
      retval = true;
    end if
  else
    retval = false;
  end if
  }
  endwhile
  return retval;
```

```
bool add(int k)
  bool restart = true, retval;
  while (restart)
  ExE p,c = locate(k);
  atomic{
  if p.next == c && !p.mark then
    restart = false;
    if c.key ≠ k then
      E t = alloc(E);
      t.mark = false;
      t.key = k;
      t.next = c;
      p.next = t;
      retval = true;
    end if
  else
    retval = false;
  end if
  }
  endwhile
  return retval;
```

Figure 11. ORVYY set [15].

6). The ordering and reversibility between a prefix sequence and single sub-operations is defined in a similar manner (lines 5,6,7). Lines 8 and 9 state that two non-overlapping operations are non-reversible.

The closure of a prefix u (Closure(u)) is the set of all prefixes v that can follow u in a valid history (lines 25-35) and the order of execution of u and v cannot be reversed. Complexity of calculating Clousure(u) is O(n^2), where n is the total number of prefixes. Our algorithm checks if for each proper prefix x, for each prefix y that can occur in-between x and next(x), the prefix y can be moved before x or after next (x) in the sequence using the trace transformation (using property of reversibility) (lines 18-24). If the check fails for any prefix $x \in \mathbf{I}$, the algorithm returns false otherwise it returns true.

The algorithm returning true means all possible valid histories (for the input specification) are equivalent to some valid sequential history with the same order of non-overlapping operations. This in turn implies linearizability of the implementation using Definition 7.

In case of failure, our checker returns a valid history (sequence of sub-operations) which cannot be mapped to a sequential history. If the check on line 19 fails for prefix x and $y \in$ Closure(x) then the failing valid history is the sequence x,...,y, next(x). The dotted sequence is a sequence of prefixes such that for any subsequence u, v u and v are in Closure(x), $u \wr v$ and $\neg u \ominus v$.

VI. Handling Complex Operation Interactions

We found that for complex concurrent operations, the ordering and reversibility of sub-operation pairs varies depending on the relative values of the involved variables. There are only finite number of ways in which two operations can interact with each other i.e., how values of involved variables can relate to each other. Linearizability of an implementation can be checked by applying our technique using all possible interactions of the involved operations. We illustrate this process using O'Hearn et al.'s Lazy set [15] (ORVYY set).

```
T1. struct E{int mark, int key, E next};

contains()
C1.   E P;
C2.   E C;
C3.   1 READ P;
C4.   1 C = P.next;
C5.   1 READ C.key;

remove()
R1.   E P;
R2.   E C;
R3.   1 READ P;
R4.   1 C = P.next;
R5.   1 < P[next] == C; P.mark == 0; >
            READ C.key;
            C.mark = 1;
            P.next = C.next;

add()
A1.   E P;
A2.   E C;
A3.   E T;
A4.   1 READ P;
A5.   1 C = P.next;
A6.   1 < P.next == C; P.mark == 0; >
            READ C.key;
            T.next = C;
            P.next = T;
```

Figure 12. ORVYY set [15] simplified specification.

Figure 11 shows the pseudo code for the ORVYY set. There are three concurrent methods *contains*, *remove*, and *add*. The specification for the ORVYY set, written in our language, is in Figure 12. There are three operations: *contains*, *remove*, and *add*. The operation *contains (key not present)* is equivalent to the operation *contains(key present)*. The operations *add (key present)* and *remove (key not present)* are trivial extensions and have been omitted for simplicity.

The interaction between operations depends on the relationships among the shared variables involved in the operations. For example, consider the interaction of operation *add* and operation *contains* from Figure 11. The *add* operations involves three variables (P_a, C_a, T_a). The *contains* operation involves two variables (P_c, C_c). The two operations can interact with each other in several ways. Each possible interaction is a result of a different relation between the involved variables. The list of possible iterations of *contains* and *add* is:

1. $P_c = P_a$ and $C_c = C_a$
2. $P_c = P_a$ and $C_c = T_a$
3. $P_c = T_a$ and $C_c = C_a$
4. P_c, C_c and P_a, C_a, T_a are not related

Contains()(P_c, C_c)-Remove()(P_r, C_r) 1. $P_c = P_r$, $C_c = C_r$ 2. $P_c = C_r$ 3. P_c, C_c and P_r, C_r are not related
Remove()(P_{r1}, C_{r1})-Remove()(P_{r2}, C_{r2}) 1. $C_{r1} = P_{r2}$ 2. P_{r1}, C_{r1} and P_{r2}, C_{r2} are not related
Add()(P_{a1}, C_{a1}, T_{a1})-Add()(P_{a2}, C_{a2}, T_{a2}) 1. $C_{a1} = P_{a2}$ 2. $P_{a1} = P_{a2}$, $C_{a1} = T_{a2}$ 3. $P_{a1} = T_{a2}$, $C_{a1} = C_{a2}$ 4. P_{a1}, C_{a1}, T_{a1} and P_{a2}, C_{a2}, T_{a2} are not related
Remove(P_r, C_r)-Add()(P_a, C_a, T_a) 1. $P_r = P_a, C_r = T_a$ 2. $P_r = T_a, C_r = C_a$ 3. $P_r = C_a$ 4. P_r, C_r and P_a, C_a, T_a are not related
Contains()(P_c, C_c) -Add()(P_a, C_a, T_a) 1. $P_c = P_a, C_c = C_a$ 2. $P_c = P_a, C_c = T_a$ 3. $P_c = T_a, C_c = C_a$ 4. P_c, C_c and P_a, C_a, T_a are not related

Figure 13. Operation interactions for the ORVYY set.

Note that other cases are either infeasible or equivalent to one of the aforementioned cases. For example, the case where $C_c = P_a$ is equivalent to case 4. All the interactions between operations for the ORVYY set are listed in Figure 13. We use multiple versions of the same operation to cover every possible combination of interactions between operations (each version is considered as a new operation). The ordering and reversibility properties of the pairs corresponding to each pair of operations are defined according to the premises. The resulting ordering and reversibility definitions are fed to the checker to verify if the implementation is linearizable.

VII. Soundness Proof

Now we show that our linearizability check is sound. First we show that for any input which passes the linearizability check, all valid histories with respect to the input specification are equivalent to some valid sequential history with the same order of non-overlapping operations. We prove this by induction over the length of valid histories, where length of history refers to number of sub-operations in the history.

Base case: A sequence consisting of a single sub-operation is a trivially valid sequential history.

Given: A history of length k maps to a valid sequential history with order of non-overlapping operations preserved.

To Prove: Any valid history formed by appending a new sub-operation to the history can also be mapped to a valid sequential history with the order of non-overlapping operations preserved.

In Figure 14 we start with a history of length $k+1$ formed by appending sub-operation S_{n+1} from operation instance S to a history of length k. The history of length k can be mapped to a valid sequential history. The valid sequential history will be a sequence of prefixes. The prefix $S_1 S_2 \dots S_n$ of operation instance S denoted by P_s will be present in the sequential history. The history formed by replacing the history of length k with its sequential counterpart falls under one of two cases:

- Case 1: P_s is immediately followed by prefix X such that $P_s \ominus X$. In this case, we reverse the order of P_s and X in the history (line 2).
- Case 2: P_s is immediately followed by X such that $\neg P_s \ominus X$ and X is immediately followed by Y such that $X \ominus Y$. In this case, we reverse the order of X and Y in the history (line 3).

We apply case 2 for prefixes down the sequence until no further order change is possible. The result is a history where P_s if followed by prefixes, each of which is an element of Closure(P_s) (line 4). Since our linearizability check guarantees that for every element Z in Closure(P_s) which can occur before S_{n+1}, $Z \ominus S_{n+1}$. Using this property, we reverse the oder of Z and S_{n+1}. The final result will be a history where sequence $P_s S_{n+1}$ is followed by a sequence of prefixes, which is a valid sequential history (line 5). The order of non-overlapping operations remains the same because the boundary pairs is always non-reversible i.e., their order cannot be changed.

Using Definition 7 we can say that any implementation which holds Assumption 1 and passes the linearizability test is linearizable with respect to the abstract data structure.

VIII. Incompleteness

Our technique is not complete, i.e., an implementation which fails the linearizability test may or may not be non-linearizable. Specifically, a linearizable implementation can fail our linearizability test for two reasons:

1. Algorithms which do not preserve internal data structure state: There are linearizable algorithms which do not preserve the state of internal data structures when mapping a history to a sequential history. An example of this case is the Herlihy-Wing queue [1]. The queue is implemented using an unbounded length array and a pointer storing the upper end of the array. Let H' be the sequential history corresponding to a concurrent history H; then the execution of H and H' may leave the array with elements at different indexes.

Since our technique conserves the state of internal data structures while mapping a history to sequential a history, it returns False for the Herlihy-Wing queue.

2. Conservative definition of history (Definition 5): A history H, as defined in Definition 5, is the superset of all possible sets of histories allowed by an implementation. Let S be the set of all possible histories categorized as valid according to Definition 5, for a given implementation. It is theoretically possible to design an implementation for which S will include histories which can never be executed. If the linearizability test fails for such histories then our technique will result in a false positive. In such a case, the non-linearizable sequence of sub-operations returned by the linearizability checker can be manually verified to be non-executable, i.e., impossible at runtime.

IX. Evaluation

We have applied our technique on a number of popular implementations of concurrent stacks, queues, and sets. Our

1.	$[\underbrace{\dots\dots\dots\dots\dots\dots}_{\text{Valid history of length } k}]S_{n+1}$	$\equiv$	$[\dots\underbrace{S_1 S_2 \dots S_n}_{\text{Prefix S}}\underbrace{\dots\dots\dots\dots\dots}_{\text{Valid sequence of prefixes}}]S_{n+1}$	Replacing history of length k by corresponding sequential history. $S_1 S_2 \dots S_n$ denoted by P_s
2.	$\dots P_s \underbrace{XY\dots\dots\dots\dots}_{\text{Valid sequence of prefixes}} S_{n+1}$	$\equiv$	$\dots XP_sY\dots\dots S_{n+1}$	if($P_s \ominus X$).
3.	$\dots P_sXY\dots\dots S_{n+1}$	$\equiv$	$\dots P_sYX\dots\dots S_{n+1}$	if($\neg P_s \ominus X$ and $X \ominus Y$).
4.	$\dots\dots P_s\dots\dots\dots S_{n+1}$	$\equiv$	$\dots\dots P_sZ_1Z_2\dots\dots Z_uS_{n+1}$	where $Z_i \in$ Closure(P_s) for $1 \leq i \leq u$
5.		$\equiv$	$\dots\dots P_sS_{n+1}Z_1Z_2\dots\dots Z_u$	$\forall Z \in$ Closure(P_s), $Z \wr S_{n+1}$ $\Rightarrow Z \ominus S_{n+1}$

Figure 14. Proving soundness of linearizability check by induction.

1	2	3	4	5
Data Structure	**Operations**	**# Sub-ops**	**Time (ms)**	**Passes Check**
MS non-blocking queue [8]	Enqueue, Dequeue(empty), Dequeue(non-empty)	4	5	Yes
MS two-lock queue [8]	Enqueue, Dequeue(empty), Dequeue(non-empty)	11	9	Yes
DGLM non-block. queue [16]	Enqueue, Dequeue(empty), Dequeue(non-empty)	4	5	Yes
Herlihy-Wing Queue [1]	Enqueue, Dequeue	5	3	No
Treiber's stack [17]	Push, Pop(empty), Pop(non-empty)	5	3	Yes
Elimination back-off stack [9]	Push(eliminating), Push(eliminated), Pop(eliminating), Pop(eliminated), Push (normal), Pop (normal)	20	14	Yes
Time-stamped Stack [18]	Push(normal), Push(eliminated), Pop(eliminating), Pop (normal)	10	8	Yes
HHLMSS Lazy set [19]	Contains, Remove(key present), Add (key not present)	23	29	Yes
VY CAS set [20]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	20	23	Yes
VY DCAS set [20]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	19	23	Yes
ORVYY set [15]	Contains, Remove(key present), Remove(key not present), Add(key present), Add(key not present)	15	19	Yes
Pair snapshot [21]	Read-pair, write	5	3	Yes
RDCSS [22]	RDCSS, RDCSS_Read, CAS_Write	5	4	Yes

Table I
CHECKING LINEARIZABILITY OF DIFFERENT CONCURRENT DATA STRUCTURE IMPLEMENTATIONS.

static checker is a C++ implementation of Algorithm 1 running on an Intel(R) Xeon(R) CPU E5607 @ 2.27GHz with 16 GB RAM, Linux kernel version 2.6.32. Table I presents our findings. For each benchmark, the table reports operations we considered for the implementation (column 2) and the total number of sub-operations across all operations (column 3). Column 4 gives the time taken by our static checker took (in milliseconds). Column 5 indicates if the benchmark passed or failed the check. We have kept the granularity of sub-operations limited to single reads, writes, and synchronization primitives. The granularity can be easily increased for trivial cases (by combining consecutive sub-operations).

A. Benchmarks

The **MS non-blocking queue** was our running example. **MS two-lock queue** is the two-lock based queue from the same paper [8]. There are two methods described in the algorithm, *enqueue* and *dequeue*. The *dequeue* method corresponds to two operations, one for the successful dequeue and the other for the empty-queue dequeue.

DGLM non-blocking queue [16] is a modified version of the MS non-blocking queue. The specification for the DGLM non-blocking queue varies from the MS non-blocking queue in terms of pre-condition for the sub-operations. The sub-operation ordering and reversibility remain the same for both the benchmarks.

Herlihy-Wing queue is an array-based queue described in the original linearizability paper [1]. The *Dequeue* method for an empty queue never terminates (that is why we have considered only the successful dequeue operation in our check). As described in Section VIII, our technique fails to prove the Herlihy-Wing queue linearizable.

Treber's stack [17] is the simplest form of a non-blocking concurrent stack algorithm. It is a linked-list based implementation, and the operations — *Push*, *Pop(empty)*, and *Pop(non-empty)* — are performed using CAS.

Elimination back-off stack [9] is an elimination-based lock-free stack. Elimination refers to canceling out concurrent push and pop operations without modifying the central data structure. The elimination process uses two auxiliary arrays. A pair of concurrently executing push and pop are eliminated. There is no sequential execution equivalent for such a case. We handled this case by distributing the sub-operations between eliminated and eliminating operations. This way there exists a sequential execution equivalent of the two eliminated operations. The implementation supports *push* and *pop* methods on the stack. The elimination parts leads to four operations: *push (eliminating)*, *push (eliminated)*, *pop (eliminating)*, and *pop (eliminated)*.

Time-stamped stack [18] is a linked-list based stack where each thread has its own linked list, using timestamps to avoid total ordering in concurrent stack operations; it also uses elimination to increase performance. The stack supports *push* and *pop* methods. The implementation has four operations, *push(normal)*, *push (eliminated)*, *pop (eliminating)*, and *pop (normal)*. We have not considered the stack empty check for this implementation. In addition, the *pop* operation is limited to setting the *deleted marker* which is associated with each node. Finally, we have not considered the removal of the node from the linked list.

HHLMSS Lazy set [19] is a linked-list based set which uses locks. Each node has a lock associated with it. The implementation supports three methods: *contains*, *remove* and *add*. The *contains* method is wait-free. The operations in the implementation that we have considered are *contains*, *remove (key present)*, and *add(key not present)*.

VY CAS set and **VY DCAS set** [20] are linked-list based set algorithms which use Compare-and-swap (CAS) and Double Compare-and-swap (DCAS) primitives for synchronization. The *contains* method is wait-free. The operations involved in the implementation are *contains*, *remove (key present)*, *remove (key not present)*, *add (key-present)* and *add(key not present)*.

ORVYY set [15] is also a linked-list based set which uses a marked bit for marking deleted nodes; it has been discussed in detail in Section VI.

Pair snapshot [21] reads two variables atomically in the presence of concurrent writes.

RDCSS [22] is an atomic multiword compare-and-swap. It works in presence of RDCSS read and CAS based writes.

B. Discussion

The evaluation shows that our technique is applicable to a variety of data structure implementations, regardless of which synchronization techniques they use. Our technique is very efficient (running time is at most 29 ms) because the static checker explores a very small search space compared to other techniques. Tools like CAVE [23] and Poling [2] take several seconds to several hundred seconds for the benchmarks they can handle. Note that we are not comparing our verification time to theirs — it would be inappropriate to do so, as their techniques are fully automatic. The main strength of our technique is that it is applicable to any concurrent data structure, i.e. it is generic. The *Time-stamped stack* has never been handled by any linearizability verification technique. The paper presenting the data structure provides a very customized linearizability proof for the algorithm. The main overhead of our technique is specifying the operations in terms of sub-operations. We found that the method of specifying operations varies with the technique used for synchronization. The elimination technique has been used in elimination back-off stack and time-stamped stack. Elimination leads to different versions of operations depending upon whether the operation is being eliminated or is eliminating. The set algorithms, the time-stamped stack, and the pair-snapshot benchmarks had complex interactions among the operations. We handled these benchmarks by using multiple versions of some operations (as described in Section VI). The MS non-blocking queue, the elimination back-off stack, and the RDCSS benchmarks had operations distributed across multiple threads.

X. Related Work

We presented a general and practical technique for checking the linearizability of concurrent data structure implementations. We now discuss other techniques used for verifying linearizability; our focus is on generic techniques that can be applied to more than one concurrent data structure.

Model-checking linearizability [24], [25] aims at exploring all possible linearization points and finding a counter example; this does not guarantee soundness. There are linearization-point based proof techniques for which the linearization points are user specified or automatically inferred by the techniques. Such techniques fail to handle more complicated algorithms where an operation's linearization point depends upon other concurrently executing operations.

Most of the techniques for proving linearizability are tied to a particular class of concurrent data structures. There are techniques which work only for the algorithms which have the linearization point inside the operation code [26]. Other techniques work for external linearization points only for read only operations [27]. Reduction based technique presented in [7] requires *moverness* which is too strong a criterion limiting the application of the approach. The backward-simulation based technique in [28] claims to be applicable to all concurrent data structure; it handles the Herlihy-Wing queue as well, which we cannot. According to the paper the authors had to write 500 proof rules in the KIV theorem prover just for the specific Herlihy-Wing queue. Authors have applied their technique only on the Herlihy-Wing queue and extending the technique to other data structures is not trivial. Liang and Feng [3] use instrumentation and rely-guarantee reasoning. The technique is specifically built to handle concurrent data structure implementations which have *helping mechanisms* and *future dependent linearization points*. Vafeiadis's approach [23] works on a number of data structures but fails in verifying complex set algorithms. Zhu et al. [2] handle data structures implementations which follow the patterns of *thread helping* and *hindsight*. In contrast to these techniques, our method is not dependent on any property of the data structure implementation. Another advantage of our technique is that when the check fails, our method provides the user with a sequence of sub-operations which cannot be linearized.

XI. Conclusion

We have presented a generic and sound technique for proving concurrent data structure implementations linearizable. We provide the user with a specification language and a static checker. Our technique is independent of any properties of the implementation in question. We have applied our technique to a number of queue, stack, and set algorithms, as well as concurrent programs. We found that writing specifications is straightforward, and the checking process is very efficient.

References

[1] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[2] H. Zhu, G. Petri, and S. Jagannathan, "Poling: Smt aided linearizability proofs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, vol. 9207, pp. 3–19. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-21668-3_1

[3] H. Liang and X. Feng, "Modular verification of linearizability with non-fixed linearization points," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 459–470. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462189

[4] L. Wang and S. D. Stoller, "Static analysis of atomicity for programs with non-blocking synchronization," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 61–71. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065953

[5] T. Elmas, S. Qadeer, and S. Tasiran, "A calculus of atomic actions," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09. New York, NY, USA: ACM, 2009, pp. 2–15. [Online]. Available: http://doi.acm.org/10.1145/1480881.1480885

[6] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, Dec. 1975. [Online]. Available: http://doi.acm.org/10.1145/361227.361234

[7] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran, "Simplifying linearizability proofs with reduction and abstraction," in *Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Verlag, April 2010. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=130595

[8] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275. [Online]. Available: http://doi.acm.org/10.1145/248052.248106

[9] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1007912.1007944

[10] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," in *Proceedings of the 34th Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, June 2013, pp. 231–242.

[11] C. Flanagan, S. N. Freund, and S. Qadeer, "Exploiting purity for atomicity," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 275–291, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.47

[12] M. Lesani, T. Millstein, and J. Palsberg, *Automatic Atomicity Verification for Clients of Concurrent Data Structures*. Cham: Springer International Publishing, 2014, pp. 550–567. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_37

[13] L. Groves, "Verifying michael and scott's lock-free queue algorithm using trace reduction," in *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, ser. CATS '08. Darlinghurst, Australia: Australian Computer Society, Inc., 2008, pp. 133–142. [Online]. Available: http://dl.acm.org/citation.cfm?id=1379361.1379385

[14] I. B. M. Corporation, *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085, 1983.

[15] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh, "Verifying linearizability with hindsight," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ser. PODC '10. New York, NY, USA: ACM, 2010, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835722

[16] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a practical lock-free queue algorithm." in *FORTE*, ser. Lecture Notes in Computer Science, D. de Frutos-Escrig and M. Núñez, Eds., vol. 3235. Springer, 2004, pp. 97–114. [Online]. Available: http://dblp.uni-trier.de/db/conf/forte/forte2004.html#DohertyGLM04

[17] R. Treiber, "Systems programming : coping with parallelism," IBM US Research Centers (Yorktown,San Jose,Almaden, US), Tech. Rep. RJ 5118, 1986. [Online]. Available: http://opac.inria.fr/record=b1015261

[18] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676963

[19] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," in *Proceedings of the 9th International Conference on Principles of Distributed Systems*, ser. OPODIS'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 3–16. [Online]. Available: http://dx.doi.org/10.1007/11795490_3

[20] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 125–135. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375598

[21] S. Qadeer, A. Sezgin, and S. Tasiran, "Back and forth: Prophecy variables for static verification of concurrent programs," Tech. Rep. MSR-TR-2009-142, October 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=103176

[22] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proceedings of the 16th International Conference on Distributed Computing*, ser. DISC '02. London, UK, UK: Springer-Verlag, 2002, pp. 265–279. [Online]. Available: http://dl.acm.org/citation.cfm?id=645959.676137

[23] V. Vafeiadis, "Modular fine-grained concurrency verification," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-726, Jul. 2008. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf

[24] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: A complete and automatic linearizability checker," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 330–340. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806634

[25] M. Vechev, E. Yahav, and G. Yorsh, "Experience with model checking linearizability," in *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 261–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02652-2_21

[26] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine, "An integrated specification and verification technique for highly concurrent data structures." in *TACAS*, ser. Lecture Notes in Computer Science, N. Piterman and S. A. Smolka, Eds., vol. 7795. Springer, 2013, pp. 324–338. [Online]. Available: http://dblp.uni-trier.de/db/conf/tacas/tacas2013.html#AbdullaHHJR13

[27] J. Derrick, G. Schellhorn, and H. Wehrheim, "Verifying linearisability with potential linearisation points," in *FM 2011: Formal Methods*, ser. Lecture Notes in Computer Science, M. Butler and W. Schulte, Eds. Springer Berlin Heidelberg, 2011, vol. 6664, pp. 323–337. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21437-0_25

[28] G. Schellhorn, H. Wehrheim, and J. Derrick, "How to prove algorithms linearisable," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. Seshia, Eds. Springer Berlin Heidelberg, 2012, vol. 7358, pp. 243–259. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_21