Comparison Under Abstraction for Verifying Linearizability

Daphna Amit[1,⋆], Noam Rinetzky[1,⋆⋆], Thomas Reps[2,⋆⋆⋆], Mooly Sagiv[1],
and Eran Yahav[3]

[1] Tel Aviv University
{amitdaph,maon,msagiv}@tau.ac.il
[2] University of Wisconsin
reps@cs.wisc.edu
[3] IBM T.J. Watson Research Center
eyahav@us.ibm.com

Abstract. *Linearizability* is one of the main correctness criteria for implementations of concurrent data structures. A data structure is *linearizable* if its operations appear to execute atomically. Verifying linearizability of concurrent unbounded linked data structures is a challenging problem because it requires correlating executions that manipulate (unbounded-size) memory states. We present a static analysis for verifying linearizability of concurrent unbounded linked data structures. The novel aspect of our approach is the ability to prove that two (unbounded-size) memory layouts of two programs are isomorphic in the presence of abstraction. A prototype implementation of the analysis verified the linearizability of several published concurrent data structures implemented by singly-linked lists.

1 Introduction

Linearizability [1] is one of the main correctness criteria for implementations of concurrent data structures (a.k.a. *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. One of the benefits of linearizability is that it simplifies reasoning about concurrent programs. If a concurrent object is linearizable, then it is possible to reason about its behavior in a concurrent program by reasoning about its behavior in a (simpler) sequential setting.

Informally, a concurrent object o is linearizable if each concurrent execution of operations on o is equivalent to some permitted sequential execution, in which the global order between non-overlapping operations is preserved. The equivalence

[⋆] Supported by a grant from the Israeli Academy of Science.

[⋆⋆] Supported in part by the German-Israeli Foundation for Scientific Research and Development (G.I.F.), and in part by a grant from the Israeli Academy of Science.

[⋆⋆⋆] Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 477–490, 2007.

is based on comparing the arguments and results of operations (responses). The permitted behavior of the concurrent object is defined in terms of a specification of the desired behavior of the object in a sequential setting.

Linearizability is a widely-used concept, and there are numerous non-automatic proofs of linearizability for concurrent objects (See Sec. 6). Proving linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution. Proving linearizability for concurrent objects that are implemented by dynamically allocated linked data-structures is particularly challenging, because it requires correlating executions that may manipulate memory states of unbounded size.

In this paper, we present a novel technique for *automatically* verifying the linearizability of concurrent objects implemented by linked data structures. Technically, we verify that a concurrent object is linearizable by simultaneously analyzing the concurrent implementation with an *executable sequential specification* (i.e., a sequential implementation). The two implementations manipulate two disjoint instances of the data structure. The analysis maintains a partial isomorphism between the memory layouts of the two instances. The abstraction is precise enough to maintain isomorphism when the difference between the memory layouts is of bounded size. Note that the memory states themselves can be of unbounded size.

Implementation. We have implemented a prototype of our approach, and used it to automatically verify the linearizability of several concurrent algorithms, including the queue algorithms of [2] and the stack algorithm of [3]. As far as we know, our approach is the first *fully automatic proof* of linearizability for these algorithms.

Limitations. Our analysis has several limitations: (i) Every concurrent operation has a (specified) *fixed linearization point*, a statement at which the operation appears to take effect. (This restriction can be relaxed to several statements, possibly with conditions.) (ii) We verify linearizability for a fixed but arbitrary number of threads. (iii) We assume a garbage collected environment. Sec. 4 discusses the role of these limitations. We note that the analysis is always sound, even if the specification of linearization points is wrong (see [4]).

Main Results. The contributions of this paper can be summarized as follows:

- We present the first fully automatic algorithm for verifying linearizability of concurrent objects implemented by unbounded linked data structures.
- We introduce a novel heap abstraction that allows an isomorphism between mutable linked data structures to be maintained under abstraction.
- We implemented our analysis and used it to verify linearizability of several unbounded linked data structures.

Due to space reasons, we concentrate on providing an extended overview of our work by applying it to verify the linearizability of a concurrent-stack algorithm due to Treiber [3]. Formal details can be found in [4].

```
[10] #define EMPTY -1
[11] typedef int data_type;
[12] typedef struct node_t {
[13]   data_type d;
[14]   struct node_t *n
[15] } Node;
[16] typedef struct stack_t {
[17]   struct node_t *Top;
[18] } Stack;
```

(a) Stack and Node type definitions

```
[40] void client(Stack *st) {
[41]   do {
[42]     if (?)
[43]       push(st, rand());
[44]     else
[45]       pop(st);
[46]   } while (1);
[47] }
```

(c) The most general client of Stack

```
[20] void push(Stack *S, data_type v){
[21]   Node *x = alloc(sizeof(Node));
[22]   x->d = v;
[23]   do {
[24]     Node *t = S->Top;
[25]     x->n = t;
[26]   } while (!CAS(&S->Top,t,x));   // @1
[27] }

[30] data_type pop(Stack *S){
[31]   do {
[32]     Node *t = S->Top;            // @2
[33]     if (t == NULL)
[34]       return EMPTY;
[35]     Node *s = t->n;
[36]   } while (!CAS(&S->Top,t,s));   // @3
[37]   data_type r = t->d;
[38]   return r;
[39] }
```

(b) Concurrent stack procedures

Fig. 1. A concurrent stack: (a) its type, (b) implementation, and (c) most general client

2 Verification Challenge

Fig. 1(a) and (b) show *C*-like pseudo code for a concurrent stack that maintains its data items in a singly-linked list of nodes, held by the stack's Top-field. Stacks can be (directly) manipulated only by the shown procedures push and pop, which have their standard meaning.

The procedures push and pop attempt to update the stack, but avoid the update and retry the operation when they observe that another thread changed Top concurrently. Technically, this is done by repeatedly executing the following code: At the beginning of every iteration, they read a local copy of the Top-field into a local variable t. At the end of every iteration, they attempt to update the stack's Top-field using the Compare-and-Swap (CAS) synchronization primitive. CAS(&S->Top,t,x) atomically compares the value of S->Top with the value of t and, if the two match, the CAS succeeds: it stores the value of x in S->Top, and evaluates to 1. Otherwise, the CAS fails: the value of S->Top remains unchanged and the CAS evaluates to 0. If the CAS fails, i.e., Top was modified concurrently, push and pop restart their respective loops.

Specification. The linearization point of push is the CAS statement in line [26] (marked with @1). This linearization point is conditional: Only a successful CAS is considered to be a linearization point. Procedure pop has two (conditional) linearization points: Reading the local copy of Top in line [32] (marked with @2) is a linearization point, if it finds that Top has a *NULL*-value. The CAS in line [36] (marked with @3) is a linearization point, if it succeeds.

Goal. We verify that the stack algorithm is linearizable with the specified linearization points for 2 threads, using its own code as a sequential specification.

3 Our Approach

We use abstract interpretation of a non-standard concrete semantics, the *correlating semantics*, abstracted by a novel *delta heap abstraction* to conservatively verify that every execution of any program that manipulates a stack using 2 threads is linearizable. Technically, we simulate the executions of all such programs using a single program that has two threads running the stack's most-general-client and using a shared stack. (The stack's most general client, shown in Fig. 1(c), is a procedure that invokes an arbitrary nondeterministic sequence of operations on the stack.)

3.1 The Correlating Semantics

The correlating semantics "checks at runtime" that an execution is linearizable. It simultaneously manipulates two memory states: the *candidate* state and the *reference* state. The *candidate* state is manipulated according to the interleaved execution. Whenever a thread reaches a linearization point in a given procedure, e.g., executes a successful `CAS` while pushing data value 4, the correlating semantics invokes the same procedure with the same arguments, e.g., invokes `push` with 4 as its value argument, on the reference state. The interleaved execution is not allowed to proceed until the execution over the reference state terminates. The reference response (return value) is saved, and compared to the response of the *corresponding* candidate operation when it terminates. This allows to directly test the linearizability of the interleaved execution by constructing a (serial) *witness* execution for every interleaved execution. In the example, we need to show that corresponding `pop`s return identical results.

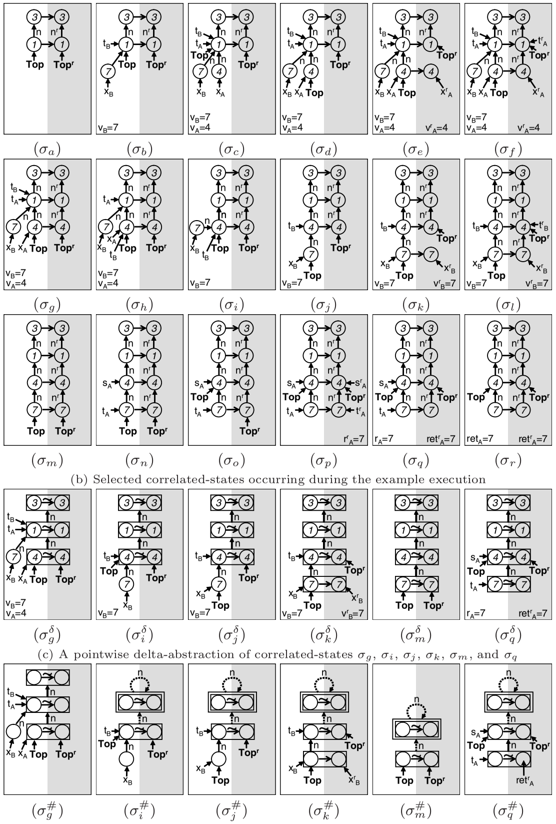
Example 1. Fig. 2(a) shows a part of a candidate execution and the corresponding fragment of the reference execution (the witness) as constructed by the correlating semantics. Fig. 2(b) shows some of the correlated states that occur in the example execution. Every correlated state consists of two states: the candidate state (shown with a clear background), and the reference state (shown with a shaded background).

The execution fragment begins in the correlated state σ_a. The candidate (resp. reference) state contains a list with two nodes, pointed to by the `Top`-*field* of the candidate (resp. reference) stack. To avoid clutter, we *do not draw the `Stack` object* itself. In the reference state we add an r-superscript to the names of fields and variables. (We subscript variable names with the id of the thread they belong to.) For now, please ignore the edges crossing the boundary between the states.

In the example execution, thread B pushes 7 into the stack, concurrently with A pushing 4. The execution begins with thread B allocating a node and linking it to the list. At this point, σ_b, thread A's invocation starts. Although B's invocation precedes A's invocation, thread A reaches a linearization point before B. Thus, after thread A executes a successful `CAS` on state σ_c, resulting in state σ_d, the correlating semantics freezes the execution in the candidate state and starts A executing `push(4)` uninterruptedly in the reference state. When

	Candidate		Reference	
Σ	A	B	A	B
σ_a		**inv push(7)**		
		[21] x=alloc()		
		[22] x->d=v		
		[24] t=S->Top		
		[25] x->n=t		
σ_b	**inv push(4)**			
	[21] x=alloc()			
	[22] x->d=v			
	[24] t=S->Top			
	[25] x->n=t			
σ_c	[26] CAS(...)			
σ_d	@1		**inv push(4)**	
			[21] x=alloc()	
			[22] x->d=v	
σ_e			[24] t=S->Top	
			[25] x->n=t	
σ_f			[26] CAS(...)	
			res push(4)	
σ_g		[26] CAS(...)		
		[24] t=S->Top		
σ_h	**res push(4)**			
		[25] x->n=t		
σ_i		[26] CAS(...)		
σ_j		@1		**inv push(7)**
				[21] x=alloc()
				[22] x->d=v
σ_k				[24] t=S->Top
				[25] x->n=t
σ_l				[26] CAS(...)
				res push(7)
		res push(7)		
σ_m	**inv pop()**			
	[32] t=S->Top			
	[33] if ( )			
	[35] s=t->n			
σ_n	[36] CAS(...)			
σ_o	@3		**inv pop()**	
			[32] t=S->Top	
			...	
			[37] r=t->d	
σ_p			[38] return r	
			res pop() : 7	
	[37] r=t->d			
σ_q	[38] return r			
σ_r	**res pop() : 7**			

(a) An example execution of the correlated semantics



(b) Selected correlated-states occurring during the example execution

(c) A pointwise delta-abstraction of correlated-states σ_g, σ_i, σ_j, σ_k, σ_m, and σ_q

(d) A pointwise bounded delta-abstraction of correlated-states σ_g, σ_i, σ_j, σ_k, σ_m, and σ_q

Fig. 2. An example correlated execution trace

the reference execution terminates, in σ_g, the candidate execution resumes. In this state, thread B has in $\mathtt{t}_B$ an old copy of the value of the stack's `Top`. Thus, its `CAS` fails. B retries: it reads the candidate's `Top` again and executes another (this time successful) `CAS` in state σ_i. Again, the correlating semantics freezes the candidate execution, and makes B execute `push(7)` on the reference state starting from σ_j. In σ_m, both `push` operations end.

Thread A invokes a `pop` operation on the stack in state σ_m. Thread A executes a successful `CAS` on state σ_n, and the reference execution starts at σ_o. When the latter terminates, the correlating semantics saves the return value, 7, in the special variable ret^r_A. When the candidate `pop` ends in σ_r, the correlating semantics stores the return value, 7, in ret_A, and compares the two, checking that the results match.

Up to this point, we described one aspect of the correlating semantics: checking that an interleaved execution is linearizable by comparing it against a (constructed) serial witness. We now show how our algorithm uses abstraction to conservatively represent unbounded states and utilizes (delta) abstraction to determine that corresponding operations have equal return values.

Comparison of Unbounded States. Our goal is to statically verify linearizability. The main challenge we face is devising a bounded abstraction of the correlating semantics that allows establishing that *every* candidate `pop` operation, in every execution, returns the same result as its corresponding reference `pop` operation. Clearly, using separated bounded abstractions of the candidate and the reference stack will not do: Even if both stacks have the same abstract value, it does not necessarily imply that they have equal contents.

Our abstraction allows one to establish that corresponding operations return equal values by using the similarity between the candidate and reference states (as can be observed in Fig. 2(b)). In particular, it maintains a mapping between the isomorphic parts of the two states (an isomorphism function). Establishing an isomorphism function—and maintaining it under mutations—is challenging. Our approach, therefore, is to incrementally construct a specific isomorphism during execution: The correlating semantics tracks pairs of nodes allocated by corresponding operations using a *correlation* relation. We say that two correlated nodes are *similar* if their `n`-successors are correlated (or both are *NULL*). The maintained isomorphism is the correlation relation between similar nodes.

Example 2. The edges crossing the boundary between the candidate and the reference component of the correlated states shown in Fig. 2(b) depict the correlation relation. In state σ_a, each node is similar to its correlated node. In states σ_b and σ_c, threads B and A have allocated nodes with data values 7 and 4, respectively, and linked them to the list. When thread A's corresponding reference operation allocates a reference node, it becomes correlated in σ_e with the candidate node that A allocated. When the reference node is linked to the list, in σ_f, the two become similar. (The node allocated by B undergoes an analogous sequence of events in σ_k and σ_l).

Comparing Return Values. The analysis needs to verify that returned values of corresponding `pop`s match. Actually, it establishes a stronger property: the returned values of corresponding pops come from correlated nodes, i.e., nodes that were allocated by corresponding pushs. Note that a node's data value, once initialized, is immutable. To simplify the presentation, and the analysis, we consider correlated nodes to also have equal data values. Our analysis tracks the nodes from which the return values are read (if this is the case) and verifies that these nodes are correlated. Sec. 4 discusses the comparison of actual data values.

Example 3. Thread A executes a pop and gets the reference return value by reading the data field of the node pointed to by $\mathtt{t}^r_A$, in σ_p. The corresponding candidate pop gets the return value by reading the data field of the node pointed to by $\mathtt{t}_A$, resulting in σ_q, with 7 being $\mathtt{r}_A$'s value. Our analysis verifies that these nodes are indeed correlated. Furthermore, consider an incorrect implementation of (concurrent) push in which the loop is removed and the CAS in line [26] is replaced by the standard pointer-update statement S->Top=x. Running our example execution with this implementation, we find that thread B manages to update Top in state σ_g (instead of failing to do so with a CAS). As a result, the candidate Top is redirected to the node that B allocated, and the current node at the top of the *candidate* stack (pushed by A) is lost. However, the node that A pushed onto the reference stack is still (eventually) in the reference stack. As a result, when it is popped from the stack, it will not be correlated with the node popped from the candidate stack. Our analysis will find this out and emit a warning.

3.2 Delta Heap Abstraction

Our abstraction summarizes an unbounded number of nodes while maintaining a partial-isomorphism between the reference state and the candidate state. The main idea is to abstract *together* the isomorphic parts of the states (comprised of pairs of correlated nodes) and to explicitly record the differences that distinguish between the states. Technically, this is performed in two abstraction steps: In the first step, we apply *delta abstraction*, which *merges* the representations of the candidate and reference states by fusing correlated nodes, losing their actual addresses. In the second step, we bound the resulting *delta memory state* into an *abstract delta memory state* using *canonical abstraction* [5], losing the exact layout of the isomorphic subgraphs while maintaining a bounded amount of information on their distinguishing differences. This abstraction works well in cases where the differences are bounded, and loses precision otherwise.

Delta Abstraction. We abstract a correlated memory state into a *delta state* by *sharing* the representation of the correlated parts. Pictorially, the delta abstraction superimposes the reference state over the candidate state. Each *pair of correlated nodes* is fused into a *duo-object.* The abstraction preserves the layout of the reference memory state by maintaining a double set of fields, candidate-fields and reference-fields, in every duo-object. Recall that a pair of correlated nodes is similar if their n-successors are correlated (or both are *NULL*). In the delta representation, the candidate-field and the reference-field of a duo-object representing similar nodes are equal. Thus, we refer to a duo-object representing a pair of similar nodes as a *uniform duo-object.*

Example 4. Fig. 2(c) depicts the delta states pertaining to some of the correlated states shown in Fig. 2(b). The delta state σ_m^δ represents σ_m. Each node in σ_m is correlated, and similar to its correlated node. A duo-object is depicted as a rectangle around a pair of correlated nodes. All the duo-objects in σ_m^δ are uniform. (This is visually indicated by the $\sim$ sign inside the rectangle.) The `n`-edge of every uniform duo-object implicitly represents the (equal) value of its $\texttt{n}^r$-edge. This is indicated graphically, by drawing the `n`-edge in the middle of the uniform duo-object. For example, the `n`-edge leaving the uniform duo-object with value 1, implicitly records the $\texttt{n}^r$-edge from the reference node with value 1 to the reference node with value 3. Note that the candidate `Top` and the reference `Top`, that point to correlated nodes in σ_m, point to the same duo-object in σ_m^δ.

The delta state σ_k^δ represents σ_k. The duo-object with data-value 7 in σ_k^δ is nonuniform; it represents the pair of nodes allocated by thread B before it links the reference node to the list. (Nonuniform duo-objects are graphically depicted without a $\sim$ sign inside the rectangle.) Note that the `n`-edge of this nonuniform duo-object is drawn on its *left*-side. The lack of a $\texttt{n}^r$-edge on the right-side indicates that the $\texttt{n}^r$-field is *NULL*.

The delta state σ_i^δ represents σ_i. The non-correlated node with data-value 7 is represented as a "regular" node.

Bounded Delta Abstraction. We abstract a delta state into a bounded-size *abstract delta state*. The main idea is to represent only a bounded number of objects in the delta state as separate (non-summary) objects in the abstract delta state, and summarize all the rest. More specifically, each uniform duo-object, nonuniform duo-object, and node which is pointed to by a variable or by a `Top`-field, is represented by a unique *abstract uniform duo-object*, *abstract nonuniform duo-object*, and *abstract node*, respectively. We represent all other uniform duo-objects, nonuniform duo-objects, and nodes, by one *uniform summary duo-object*, one *nonuniform summary duo-object*, and one *summary node*, respectively. We conservatively record the values of pointer fields, and abstract away values of data fields. (Note, however, that by our simplifying assumption, every duo-object represents nodes with equal data values.)

Example 5. Fig. 2(d) depicts the abstract delta states pertaining to the delta states shown in Fig. 2(c). The abstract state $\sigma_i^\sharp$ represents σ_i^δ. The duo-objects with data values 1 and 3 in σ_i^δ are represented by the summary duo-object, depicted with a double frame. The duo-object u with data value 4 in σ_i^δ is represented by its own abstract duo-object in $\sigma_i^\sharp$ (and not by the summary duo-object) because u is pointed to by (both) `Top`-fields. The non-correlated node w with data-value 7 in σ_i^δ is pointed to by $\texttt{x}_B$. It is represented by its own abstract node pointed to by $\texttt{x}_B$. The `n`-field between the candidate node w and the duo-object u in σ_i^δ is represented in the abstract state by the solid n-labeled edge. The absence of an n-labeled edge between abstract nodes or abstract duo-objects represents the absence of pointer fields. Finally, the dotted edges represent loss of information in the abstraction, i.e., pointer fields which may or may not exist. Note that the summary duo-object in $\sigma_i^\sharp$ is uniform. This information is key to our analysis: it records the fact that the candidate and reference states have (potentially unbounded-sized) isomorphic subgraphs.

The abstract delta state $\sigma_k^\sharp$ represents σ_k^δ. The nonuniform duo-object v in σ_k^δ is represented by an abstract nonuniform duo-object in $\sigma_k^\sharp$. Note that the abstraction maintains the information that the duo-object pointed to by v's candidate `n`-field, is also pointed to by the reference `Top`. This allows to establish that once thread B links the reference node to the list, the abstract nonuniform duo-object v is turned into a uniform duo-object.

Recap. The delta representation of the memory states, enabled by the novel use of similarity and duo-objects, essentially records isomorphism of subgraphs in a *local* way. Also, it helps *simplify* other elements of the abstraction: the essence of our bounded abstraction is to keep distinct (i.e., not to represent by a summary node or a summary duo-object) nodes and pairs of correlated nodes which are pointed-to by variables or by a `Top`-field. Furthermore, by representing the reference edges of similar nodes by the candidate edges and the similarity information recorded in (uniform) duo-objects, the bounded abstraction can maintain only a single set of edges for these nodes. Specifically, if there is a bounded number of differences between the memories, the bounded abstraction is, essentially, abstracting a singly-linked list of duo-objects, with a bounded number of additional edges. In addition, to represent precisely the differences between the states using this abstraction, these differences have to be bounded, i.e., every non-similar or uncorrelated node has to be pointed to by a variable or by a `Top`-field.

Example 6. The information maintained by the abstract delta state suffices to establish the linearizability of the stack algorithm. Consider key points in our example trace:

- When thread B performs a `CAS` on σ_g, its abstraction $\sigma_g^\sharp$ carries enough information to show that it fails, and when B tries to reperform the `CAS` on σ_i, its abstraction $\sigma_i^\sharp$ can establish that the `CAS` definitely succeeds.
- When linking the reference node to the list in state σ_e and later in σ_k, the abstracted states can show that newly correlated nodes become similar.
- $\sigma_m^\sharp$, the abstraction of σ_m, which occurs when no thread manipulates the stack, indicates that the candidate and the reference stacks are isomorphic.
- Finally, $\sigma_q^\sharp$, the abstraction of σ_q, indicates that the return value of the reference `pop` was read from a node correlated to the one from which $\mathtt{r}_A$'s value was read (indicated by $\mathtt{ret}_A^r$ pointing into the correlated node). This allows our analysis to verify that the return values of both `pop`s agree.

Our analysis is able to verify the linearizability of the stack. Note that the abstraction does not record any particular properties of the list, e.g., reachability from variables, cyclicly, sharing, etc. Thus, the summary duo-object might represent a cyclic list, a shared list, or even multiple unreachable lists of duo-objects. Nevertheless, we know that the uniform summary duo-object represents an (unbounded-size) isomorphic part of the candidate and reference states.

4 Discussion

In this section, we shortly discuss some key issues in our analysis.

Soundness. The soundness of the analysis requires that every operation of the executable sequential specification is fault-free and always terminates. This ensures that triggering a reference operation never prevents the analysis from further exploring its candidate execution path. Our analysis conservatively verifies the first requirement in situ. The second requirement can be proved using termination analysis, e.g., [6]. Once the above requirements are established, the soundness of the abstract interpretation follows from the soundness of [5]'s framework for program analysis, in which our analysis is encoded. We note that for many of our benchmarks, showing termination is rather immediate because the procedures perform a loop until a `CAS` statement succeeds; in a serial setting, a `CAS` always succeeds.

Correlating Function. We used the same correlation function in all of our benchmarks: nodes allocated by corresponding operations are correlated. (In all our benchmarks, every operation allocates at most one object. More complicated algorithms might require more sophistication.) We note that *our analysis is sound with any correlation function.*

Comparison of Return Values. We simplified the example by not tracking actual data values. We now show how return values can be tracked by the analysis. The flow of data values *within corresponding operations* can be tracked from the pushed value parameter to the data fields of the allocated nodes (recall that corresponding operations are invoked with the same parameters). We then can record data-similarity, in addition to successor-similarity, and verify that data-fields remain immutable. This allows to automatically detect that return values (read from correlated nodes) are equal. Such an analysis can be carried out using, e.g., the methods of [7].

Precision. As far as we know, we present the first shape analysis capable of maintaining isomorphism between (unbounded-size) memory states. We attribute the success of the analysis to the fact that in the programs we analyze the memory layouts we compare only "differ a little". The analysis tolerates local perturbations (introduced, e.g., by interleaved operations) by maintaining a precise account of the difference (*delta*) between the memory states. In particular, during our analysis, it is always the case that every abstract object is pointed to by a variable or a field of the concurrent object, except, possibly, uniform duo-objects. Thus, we do not actually expect to summarize nonuniform duo-objects or regular nodes. In case the analysis fails to verify the linearizability of the concurrent implementation, its precision may be improved by refining the abstraction.

Operational Specification. We can verify the concurrent implementation against a *simple* sequential specification instead of its own code. For example, in the *operational specification* of `push` and `pop`, we can remove the loop and replace the `CAS` statement with a (more natural) pointer-update statement. Verifying a code against a specification, and not against itself, can improve performance.

For example, we were not able to verify a sorted-set example using its own code as a specification (due to state explosion), but we were able to verify it using a simpler specification. Also, it should be much easier to prove fault-freedom and termination for a simplified specification.

Parametric Shape Abstraction. We match the shape abstraction to the way the operations of the concurrent objects traverse the heap: When the traversal is limited to a bounded number of links from the fields of the concurrent object, e.g., stacks and queues, we base the abstraction on the values of variables. When the traversal is potentially unbounded, e.g., a sorted set, we also record sharing and reachability.

Automation. In the stack example, we used a very simple abstraction. In other cases, we had to refine the abstraction. For example, when analyzing the nonblocking-queue [2], we found it necessary to also record explicitly the successor of the tail. Currently, we refine the abstraction manually. However, it is possible to automate this process using the methods of [8]. We define the abstract transformers by only specifying the concrete (delta) semantics. The abstract effect of statements on the additional information, e.g., reachability, is derived automatically using the methods of [9]. The latter can also be used to derive the delta operational semantics from the correlating operational semantics.

Limitations. We now shortly discuss the reasons for the imposed limitations.

Fixed Linearization Points. Specifying the linearization points of a procedure using its own statements simplifies the triggering of reference operations when linearization points are reached. In addition, it ensures that there is only one (prefix of a) sequential execution corresponding to every (prefix of a) concurrent execution. This allows us to represent only one reference data structure. Extending our approach to handle more complex specification of linearization points, e.g., when the linearization point occurs in the body of another method, is a matter of future investigation.

Bounded Number of Threads. The current analysis verifies linearizability for a fixed (but arbitrary) number k of threads. However, our goal is not to develop a *parametric* analysis, but to lift our analysis to analyze an unbounded number of threads using the techniques of Yahav [10].

No Explicit Memory Deallocation. We do not handle the problem of using (dangling) references to reclaimed memory locations, and assume that memory is automatically reclaimed (garbage collected). Dangling references can cause subtle linearizability errors because of the ABA problem.[1] Our model is simplified by forbidding explicit memory deallocation. This simplifying assumption guar-

[1] The ABA problem occurs when a thread reads a value v from a shared location (e.g., `Top`) and then other threads change the location to a different value, say u, and then back to v again. Later, when the original thread checks the location, e.g., using read or CAS, the comparison succeeds, and the thread erroneously proceeds under the assumption that the location has not changed since the thread read it earlier [11].

Table 1. Experimental results. Time is measured in seconds. Experiments performed on a machine with a 3.8 Ghz Xeon processor and 4 Gb memory running version 4 of the RedHat Linux operating system with Java 5.0, using a 1.5 Gb heap.

Client type	(a) General client			(b) Producers / Consumers		
Data Structure	Threads	Time	# States	Threads	Time	# States
Stack [3]	3	555	64,618	2/2	1,432	82,497
Nonblocking queue [2]	2	1,874	116,902	1/1	15	2,518
Nonblocking queue [15]	2	340	34,611	1/1	12	1,440
Two-lock queue [2]	4	1,296	115,456	3/3	4,596	178,180
Pessimistic set [16]	2	14,153	229,380	1/1	2,981	51,755

antees that the *ABA* problem does not occur, and hence need not be treated in the model. We believe that our approach can be extended to support explicit memory deallocation, as done, e.g., in [12]. In our analysis, we do not model the garbage collector, and never reclaim garbage.

5 Implementation and Experimental Results

We have implemented a prototype of our analysis using the TVLA/3VMC [13,10] framework. Tab. 1 summarizes the verified data structures, the running times, and the number of configurations. Our system does not support automatic partial-order reductions (see, e.g., [14]). For efficiency, we manually combined sequences of thread-local statements into atomic blocks.

The stack benchmark is our running example. We analyze two variants of the well-known nonblocking queue algorithm of Michael and Scott: the original algorithm [2], and a slightly optimized version [15]. The two-lock queue [2] uses two locks: one for the head-pointer and one for the tail-pointer. The limited concurrency makes it our most scalable benchmark. The pessimistic set [16] is implemented as a sorted linked list. It uses *fine-grained locking*: Every node has its own lock. Locks are acquired and released in a "hand-over-hand" order; the next lock in the sequence is acquired before the previous one is released.

We performed our experiments in two settings: (a) every thread executes the most general client and (b) every thread is either a *producer*, repeatedly adding elements into the data structure, or a *consumer*, repeatedly removing elements. (The second setting is suitable when verifying linearizability for applications which can be shown to use the concurrent object in this restricted way.) Our analysis verified that the data structures shown in Tab. 1 are linearizable, for the number of threads listed (e.g., for the stack, we were able to verify linearizability for 4 threads: 2 producer threads and 2 consumer threads, and for 3 threads running general clients).

We also performed some *mutation experiments*, in which we slightly mutated the data-structure code, e.g., replacing the stack's `CAS` with standard pointer-field assignment, and specified the wrong linearization point. In all of these cases, our analysis reported that the data structure may not be linearizable. (See [4].)

6 Related Work

This section reviews some closely related work. For additional discussion, see [4].

Conjoined Exploration. Our approach for conjoining an interleaved execution with a sequential execution is inspired by Flanagan's algorithm for verifying commit-atomicity of concurrent objects in bounded-state systems [17]. His algorithm explicitly represents the candidate and the reference memory state. It verifies that at *quiescent points* of the run, i.e., points that do not lie between the invocation and the response of any thread, the two memory states completely match. Our algorithm, on the other hand, utilizes abstraction to conservatively represent an unbounded number of states (of unbounded size) and utilizes (delta) abstraction to determine that corresponding operations have equal return values.

Automatic Verification. Wang and Stoller [18] present a static analysis that verifies linearizability (for an unbounded number of threads) using a two-step approach: first show that the concurrent implementation executed sequentially satisfies the sequential specification, and then show that procedures are atomic. Their analysis establishes atomicity based primarily on the way synchronization primitives are used, e.g., compare-and-swap, and on a specific coding style. (It also uses a preliminary analysis to determine thread-unique references.) If a program does not follow their conventions, it has to be rewritten. (The linearizability of the original program is manually proven using the linearizability of the modified program.) It was used to derive manually the linearizability of several algorithms including the nonblocking queue of [2], which had to be rewritten. We automatically verify linearizability for a bounded number of threads. Yahav and Sagiv [12] automatically verify certain safety properties listed in [2] of the nonblocking queue and the two-lock queue given there. These properties do not imply linearizability. We provide a direct proof of linearizability.

Semi-Automatic Verification. In [15,19,20], the *PVS* theorem prover is used for a semi-automatic verification of linearizability.

Manual Verification. Vafeiadis *et. al.* [16] manually verify linearizability of list algorithms using rely-guarantee reasoning. Herlihy and Wing [1] present a methodology for verifying linearizability by defining a function that maps every state of the concurrent object to the set of all possible *abstract values* representing it. (The state can be instrumented with properties of the execution trace). Both techniques do not require fixed linearization points.

Acknowledgments. We are grateful for the comments of A. Gotsman, T. Lev-Ami, A. Loginov, R. Manevich, M. Parkinson, V. Vafeiadis, and M. Vechev.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. Trans. on Prog. Lang. and Syst. 12(3) (1990)
2. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC (1996)

3. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
4. Amit, D.: Comparison under abstraction for verifying linearizability. Master's thesis, Tel Aviv University (2007) Available at `http://www.cs.tau.ac.il/~amitdaph`
5. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. Trans. on Prog. Lang. and Syst. (2002)
6. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
7. Gopan, D., DiMaio, F., Dor, N., Reps, T.W., Sagiv, S.: Numeric domains with summarized dimensions. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, Springer, Heidelberg (2004)
8. Loginov, A., Reps, T.W., Sagiv, M.: Abstraction refinement via inductive learning. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
9. Reps, T., Sagiv, M., Loginov, A.: Finite Differencing of Logical Formulas for Static Analysis. In: Degano, P. (ed.) ESOP 2003 and ETAPS 2003. LNCS, vol. 2618, Springer, Heidelberg (2003)
10. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: POPL (2001)
11. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6) (2004)
12. Yahav, E., Sagiv, M.: Automatically verifying concurrent queue algorithms. In: Electronic Notes in Theoretical Computer Science, vol. 89, Elsevier, Amsterdam (2003)
13. Lev-Ami, T., Sagiv, M.: TVLA: A framework for Kleene based static analysis. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, Springer, Heidelberg (2000)
14. Clarke, E.M., Grumberg, J., Peled, O.: Model checking. MIT Press, Cambridge, MA, USA (1999)
15. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, Springer, Heidelberg (2004)
16. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP (2006)
17. Flanagan, C.: Verifying commit-atomicity using model-checking. In: Graf, S., Mounier, L. (eds.) Model Checking Software. LNCS, vol. 2989, Springer, Heidelberg (2004)
18. Wang, L., Stoller, S.D.: Static analysis of atomicity for programs with non-blocking synchronization. In: PPOPP (2005)
19. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
20. Gao, H., Hesselink, W.H.: A formal reduction for lock-free parallel algorithms. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, Springer, Heidelberg (2004)