Comparison under Abstraction for Verifying Linearizability

Daphna Amit
School of Computer Science, Tel-Aviv University, Israel
February 2007

Acknowledgements

I would like to extend my gratitude to all those who contributed to this work.

First and foremost, to my supervisor, Mooly Sagiv, for his insightful and dedicated guidance, his kind encouragement and support, and for maintaining a patient and positive attitude all along the way. Working with him has been a privilege, and a great introduction to research work.

A special thanks to Noam Rinetzky for his supremely dedicated consultation and for his profound contribution to every aspect of this work. This work would not have been possible without his fruitful ideas, great efforts, constant support, optimism, and unique willingness to help in any matter.

To Eran Yahav for his invaluable contribution to the writing and presentation of this work, for his assistance with 3VMC, and for his help and advice along the way.

To Thomas Reps for his insightful consultation and for carefully reviewing earlier drafts of this work.

To Roman Manevich and Tal Lev-Ami for many fruitful discussions, for their assistance with TVLA, and for reviewing an earlier draft of this work.

To Yehuda Afek, Nir Shavit and Ori Shalev for their helpful advice.

To Guy Gueta for his patient assistance and advice.

To Zur Izhakian and Shachar Rubinstein for their assistance, companionship and moral support.

To Alexey Gotsman, Alexey Loginov, Matthew Parkinson, Viktor Vafeiadis, and Martin Vechev for reviewing an earlier draft of this work.

To the Israeli Academy of Science, for their generous financial support.

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.

Contents

Chapter 1

Introduction

Linearizability [HW90] 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 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 al-

located 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 [MS96] and the stack algorithm of [Tre86]. 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 Appendix).

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.

For readability, 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 [Tre86]. Formal details can be found in the appendices.

Chapter 2

Verification Challenge

Fig. 2.1(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.

```
[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

Figure 2.1: A concurrent stack: (a) its type, (b) implementation, and (c) most general client.

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.

Verification 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.

Chapter 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. 2.1(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.

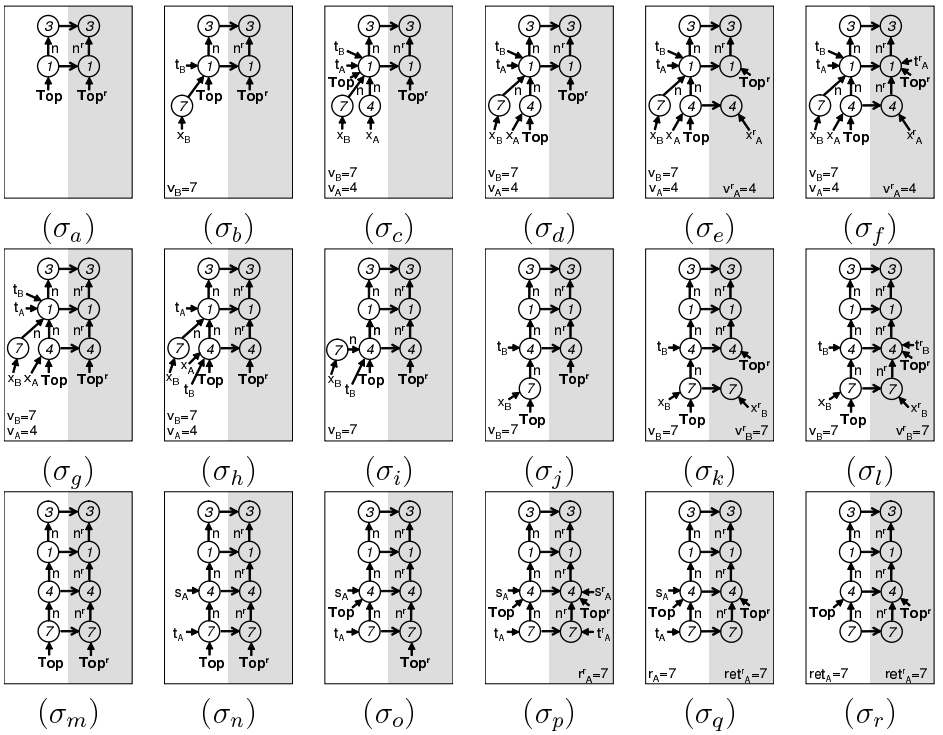
Example 3.1.1 Fig. 3.1(a) shows a part of a candidate execution and the corresponding fragment of the reference execution (the witness) as constructed by the correlating semantics. Fig. 3.1(b) shows some of the correlated states that occur in the example execution. Every correlated state consists of 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.

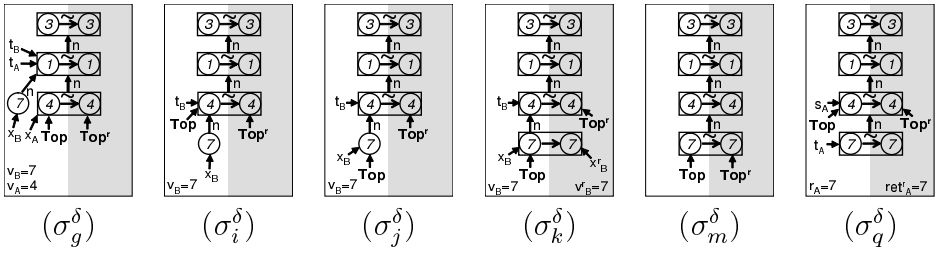
In the example execution, thread B pushes 7 into the stack, concurrently with A pushing 4. The execution begins with thread B allocating a node and linking it to the list. At this point, σ_b, thread A's invocation starts. Although B's invocation precedes A's invocation, thread A reaches a linearization 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 the reference execution terminates, in σ_g, the candidate execution resumes. In

	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 [24] t=S->Top [25] x->n=t [26] CAS(...)	
σ_e			**res push(4)**	
σ_f				
σ_g		[26] CAS(...) [24] t=S->Top		
σ_h	**res push(4)**			
σ_i		[25] x->n=t [26] CAS(...)		
σ_j		@1		**inv push(7)** [21] x=alloc() [22] x->d=v [24] t=S->Top [25] x->n=t [26] CAS(...) **res push(7)**
σ_k				
σ_l				
σ_m		**res push(7)** **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 [38] return r **res pop() : 7**	
σ_p				
σ_q	[37] r=t->d [38] return r			
σ_r	**res pop() : 7**			

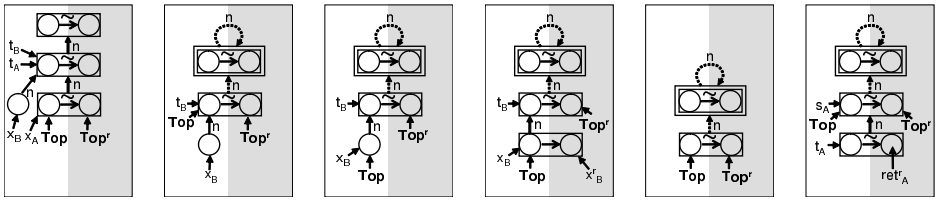
(a) An example execution of the correlated semantics



(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

Figure 3.1: An example correlated execution trace.

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 refer-

ence states (as can be observed in Fig. 3.1(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 3.1.2** The edges crossing the boundary between the candidate and the reference component of the correlated states shown in Fig. 3.1(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 `pop`s come from correlated nodes, i.e., nodes that were allocated by corresponding `push`s. 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 3.1.3 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* [SRW02], 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 3.2.1** Fig. 3.1(c) depicts the delta states pertaining to some of the correlated states shown in Fig. 3.1(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 $\mathtt{n}^r$-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 $\mathtt{n}^r$-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 3.2.2 Fig. 3.1(d) depicts the abstract delta states pertaining to the delta states shown in Fig. 3.1(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 $\mathtt{x}_B$. It is represented by its own abstract node pointed to by $\mathtt{x}_B$. 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. Specif-

ically, 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 3.2.3 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.

Chapter 4

Discussion

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

4.1 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., [BCDO06]. Once the above requirements are established, the soundness of the abstract interpretation follows from the soundness of [SRW02]'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 [GDD+04].

4.2 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.

4.3 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.

4.4 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.

4.5 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 [MS96], 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 [LRS05]. We define the abstract transformers by only specifying the concrete (delta) semantics. The abstract effect of statements on the additional information, e.g., reacha-

bility, is derived automatically using the methods of [RSL03]. The latter can also be used to derive the delta operational semantics from the correlating operational semantics.

4.6 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. (See App. A.1 for our treatment of a linearization point that depends on the future execution of the same thread.)

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 [Yah01].

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 guarantees 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 [YS03]. In our analysis, we do not model the garbage collector, and never reclaim garbage.

[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 [Mic04]. Suppose, for example, that the thread is performing a `pop`, and v is its local copy of the value of `Top`. Assume that the thread has managed to read the address w of the successor of the node that v points to before being preempted. The danger is that, when rescheduled, `pop` might continue as if w is the current successor, although it is not necessarily the case.

Chapter 5

Implementation and Experimental Results

We have implemented a prototype of our analysis using the TVLA/3VMC [LAS00, Yah01] framework. Tab. 5.1 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., [EMCGP99]). 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 [MS96], and a slightly optimized version [DGLM04]. The two-lock queue [MS96] 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 [VHHS06] 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. (See App. A for details.)

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

Client type	(a) General client			(b) Producers / Consumers		
Data Structure	Threads	Time	# States	Threads	Time	# States
Stack [Tre86]	3	555	64,618	2/2	1,432	82,497
Nonblocking queue [MS96]	2	1,874	116,902	1/1	15	2,518
Nonblocking queue [DGLM04]	2	340	34,611	1/1	12	1,440
Two-lock queue [MS96]	4	1,296	115,456	3/3	4,596	178,180
Pessimistic set [VHHS06]	2	14,153	229,380	1/1	2,981	51,755

Table 5.1: 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.

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. 5.1 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 App. A.2.)

Chapter 6

Related Work

This section reviews some closely related work.

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 [Fla04]. 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 [WS05] 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 [MS96], which had to be rewritten. We automatically verify linearizability for a bounded number of threads. Yahav and Sagiv [YS03] automatically verify certain safety properties listed in [MS96] 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

Doherty *et. al.* [DGLM04], Colvin *et. al.* [CGLM06], and Gao *et. al.* [GH04] use the *PVS* theorem prover for a semi-automatic verification of linearizability. The proof of Doherty *et. al.* is based on formalizing both the concurrent algorithm and its sequential specification as I/O Automata and showing that there exists a *simulation* relation between states of these automata. The existence of a simulation implies that every external behavior of the algorithm automaton is allowed by the specification automaton.

Manual Verification

Vafeiadis *et. al.* [VHHS06] manually verify linearizability of list algorithms using rely-guarantee reasoning. Herlihy and Wing [HW90] 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.

Dynamic Verification

Wing and Gong [WG93] *test* linearizability using executable sequential specifications by comparing a concurrent execution with every possible serial execution that agrees with the global order of operations.

Elmas et. al. [ETQ05] test linearizability using executable sequential specifications and user-specified fixed linearization points. Their tool was able to detect concurrency bugs in some industrial-scale concurrent data structure implementations.

Delta Abstraction

In some sense, the use of duo-objects reduces the problem of tracking correlated objects to that of tracking variation in related relations. Another previous use of this technique in shape analysis is the two vocabulary structures of Jeannet *et. al.* [JLRS04].

Chapter 7

Conclusions

We present an analysis that verifies linearizability by comparing under abstraction two unbounded memory states. Being able to perform such a comparison under abstraction successfully is rather surprising. When our analysis succeeds, we attribute its success to the following observation: The candidate data structure and the reference data structure are being manipulated by the same sequence of operations and return the same sequence of results. Thus, to the external observer, it seems as if the two data structures are in the same "logical state". Both data structures are represented in the same way. Thus, it is reasonable to expect that their memory layouts would be resemblant. (The only exception is the pessimistic set benchmark, where the representations of the two data structures differ. See App. A.3 for details.) By allowing the memory layouts to "differ a little", we allow the analysis to tolerate local perturbations introduced by interleaved operations.

Another key factor to the success of the analysis is that while a sequence of operations may, in general, radically change the contents of a data structure, the changes in the data structures that we analyzed are performed in small increments, and the analysis was able to track that the resemblance is preserved throughout the sequence. The delta abstraction is biased towards recording the "difference" between nearly-isomorphic memory states, which isolates the isomorphic parts of the memory states from the non-isomorphic

parts, so that the latter can be abstracted more precisely.

Bibliography

[BCDO06] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, 2006.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Inf.*, 9, 1977.

[CGLM06] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, 2006.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, 2004.

[EMCGP99] Jr. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[ETQ05] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, 2005.

[Fla04] C. Flanagan. Verifying commit-atomicity using model-checking. In *SPIN*, 2004.

[GDD+04] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, 2004.

[GH04] H. Gao and W. H. Hesselink. A formal reduction for lock-free parallel algorithms. In *CAV*, 2004.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *Trans. on Prog. Lang. and Syst.*, 12(3), 1990.

[JLRS04] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *SAS*, 2004.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS*, 2000.

[LRS05] A. Loginov, T. W. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *CAV*, 2005.

[Mic04] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6), 2004.

[MS77] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Halsted Press, New York, NY, USA, 1977.

[MS96] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[Rey02] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, 2002.

[RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, 2003.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, APR 1986.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.

[WG93] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2), 1993.

[WS05] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*, 2005.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, January 2001.

[YS03] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

List of Figures

List of Tables

Appendix A

Benchmarks

In this section, we describe the benchmarks (Sec. A.1), summarize their mutation experiments (Sec. A.2) and discuss some key points of their analysis (Sec. A.3).

A.1 General Description

Fig. A.1, Fig. A.2 and Fig. A.3 show *C*-like pseudo code for the nonblocking queue, the two-lock queue and the pessimistic set benchmarks, respectively.

Nonblocking Queue

The nonblocking queue of [MS96] (Fig. A.1) is implemented as a singly-linked list with `Head` and `Tail` pointers, pointing to the head and tail of the queue, respectively. `Head` always points to a dummy node. The procedures `enqueue` and `dequeue` iteratively attempt to update the queue without being interrupted by other threads. To obtain consistent values of various pointers, *rechecking* is used to make sure the value of a shared field hasn't changed since its local copy was read (lines `[26]` and `[45]`). The `CAS` primitive is used to update shared fields.

During interleaved execution, the `Tail` pointer may lag behind the end

of the list if an enqueuing thread has appended its new node to the list (line [28]) but has not swung Tail to point to it (line [33]). The algorithm employs a *helping mechanism*, allowing concurrently-executing threads to advance the Tail pointer (lines [31] and [49]). As a result, Tail always points to either the last or second to last node in the list.

Optimized Version

Doherty *et. al.* [DGLM04] present an optimized version of the dequeue procedure. In their version, the Tail pointer is read (and possibly advanced) only *after* Head has been updated.

Specification of Linearization Points

The linearization point of enqueue is the CAS statement in line [28] (marked with @1). This linearization point is conditional: Only a successful CAS is considered to be a linearization point. The linearization point of dequeue is more subtle: The procedure has two conditional linearization points. The CAS in line [52] (marked with @3) is a linearization point, if it succeeds. The reading of h->n in line [44] (marked with @2) is a linearization point, if the procedure returns EMPTY (i.e., executes line [48]) at the end of *the same iteration*.

Linearization point @2 is *future dependent*, since at the time line [44] is executed, it is not known how the iteration will terminate. This means that the correlating semantics cannot deterministically determine whether to trigger a reference dequeue operation after the execution of line [44] by a candidate dequeue operation. We handle this problem by triggering an auxiliary *query* operation at this point, that checks whether the reference queue is empty. The query does not alter the reference queue; it only returns a boolean value recorded by the correlating semantics. Now there are two cases to consider:

- If the candidate dequeue operation returns EMPTY at the end of the

iteration, then the last execution of line [44] is indeed a linearization point. Although the reference dequeue operation was not executed at that point, the result of the query indicates what its outcome would have been: If the reference queue was empty at the time, the operation would have left the queue unchanged and returned EMPTY. In this case, the execution of the query was equivalent to the execution of the reference operation, and the return values of the candidate and the reference operations match. Otherwise, the operation would have removed the first node in the queue and returned its data value. In this case, the return values of the candidate and the reference operations do not match, and a linearizability violation is reported.

- If the candidate dequeue operation does not return EMPTY at the end of the iteration, then the last execution of line [44] is not a linearization point. In this case, the result of the query is simply ignored, and no harm is done.

The specification of linearization points for the optimized version of dequeue is similar.

Two-Lock Queue

The two-lock queue (Fig. A.2) is implemented as a singly-linked list with Head and Tail pointers. Head always points to a dummy node. The algorithm employs two separate locks, HLock and TLock, to synchronize access to the Head and Tail pointers, respectively. Since enqueuers never have to access Head and dequeuers never have to access Tail, this allows concurrent enqueuing and dequeuing of elements.

Specification of Linearization Points

The linearization point of enqueue is line [35] (marked with @1), where the new node is appended to the tail of the queue. The linearization point

of `dequeue` is line `[43]` (marked with @2), where the successor of the head dummy is read.

Pessimistic Set

The set (Fig. A.3) is implemented as a sorted singly-linked list with a `Head` pointer. The first and last nodes in the list are dummy nodes that hold the minimum and maximum integer values, respectively. (It is assumed that all integer arguments passed to `add` and `remove` are strictly between these two values.) The intermediate nodes store the elements of the set in ascending order.

The algorithm employs fine-grained locking synchronization: Every node is associated with a lock that synchronizes access to its fields. The procedures `add` and `remove` traverse the list in a "hand-over-hand" manner, releasing a node only after its successor has been locked. This technique is also known as *lock coupling* [BS77].

Specification of Linearization Points

The linearization point of `add` is the `unlock` statement in line `[49]` (marked with @1). Intuitively, this is the point where the effect of the operation first becomes visible to other threads. Similarly, the linearization point of `remove` is the `unlock` statement in line `[77]` (marked with @2).

```
[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 queue_t {
[17]   struct node_t *Head;
[18]   struct node_t *Tail;
[19] } Queue;
```

(a) `Queue` and `Node` type definitions

```
[60] void client(Queue *q) {
[61]   do {
[62]     if (?)
[63]       enqueue(q, rand());
[64]     else
[65]       dequeue(q);
[66]   } while (1);
[67] }
```

(c) The most general client of `Queue`

```
[40] data_type dequeue(Queue *Q){
[41]   do {
[42]     Node *h = Q->Head;
[43]     Node *s = h->n;                  // @2
[44]     if (h == Q->Head)
[45]       if (s == NULL)
[46]         return EMPTY;
[47]       else
[48]         data_type r = s->d;
[49]         if (CAS(&Q->Head,h,s))       // @3
[50]           Node *t = Q->Tail;
[51]           if (h == t)
[52]             CAS(&Q->Tail,t,s);
[53]           return r;
[54]   } while (1);
[55] }
```

(d) Optimized `dequeue` procedure

```
[70] void initialize(Queue *Q) {
[71]   Node *dummy = alloc(sizeof(Node));
[72]   Q->Head = dummy;
[73]   Q->Tail = dummy;
[74] }

[20] void enqueue(Queue *Q, data_type v){
[21]   Node *x = alloc(sizeof(Node));
[22]   x->d = v;
[23]   do {
[24]     Node *t = Q->Tail;
[25]     Node *s = t->n;
[26]     if (t == Q->Tail)
[27]       if (s == NULL)
[28]         if (CAS(&t->n,s,x))          // @1
[29]           break;
[30]       else
[31]         CAS(&Q->Tail,t,s);
[32]   } while (1);
[33]   CAS(&Q->Tail,t,x);
[34] }

[40] data_type dequeue(Queue *Q){
[41]   do {
[42]     Node *h = Q->Head;
[43]     Node *t = Q->Tail;
[44]     Node *s = h->n;                  // @2
[45]     if (h == Q->Head)
[46]       if (h == t)
[47]         if (s == NULL)
[48]           return EMPTY;
[49]         CAS(&Q->Tail,t,s);
[50]       else
[51]         data_type r = s->d;
[52]         if (CAS(&Q->Head,h,s))       // @3
[53]           return r;
[54]   } while (1);
[55] }
```

(b) Concurrent queue procedures

Figure A.1: Nonblocking queue [MS96]: (a) type definitions; (b) implementation; (c) most general client; (d) optimized implementation of `dequeue` [DGLM04].

```
[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 queue_t {
[17]   struct node_t *Head;
[18]   struct node_t *Tail;
[19]   lock_type HLock;
[20]   lock_type TLock;
[21] } Queue;
```

(a) `Queue` and `Node` type definitions

```
[60] void client(Queue *q) {
[61]   do {
[62]     if (?)
[63]       enqueue(q, rand());
[64]     else
[65]       dequeue(q);
[66]   } while (1);
[67] }
```

(c) The most general client of `Queue`

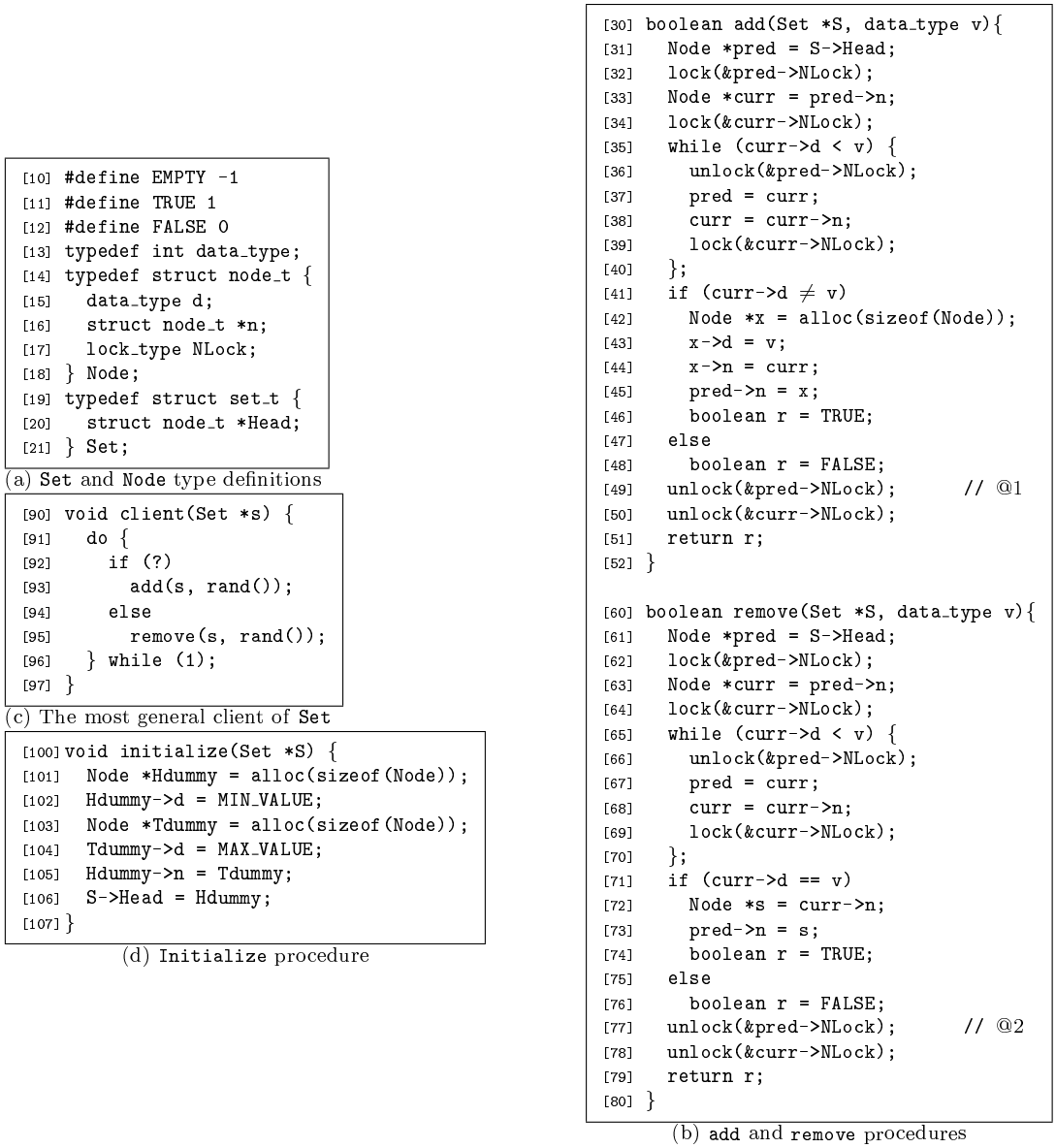
```
[70] void initialize(Queue *Q) {
[71]   Node *dummy = alloc(sizeof(Node));
[72]   Q->Head = dummy;
[73]   Q->Tail = dummy;
[74]   Q->HLock = FREE;
[75]   Q->TLock = FREE;
[76] }

[30] void enqueue(Queue *Q, data_type v){
[31]   Node *x = alloc(sizeof(Node));
[32]   x->d = v;
[33]   lock(&Q->TLock);
[34]     Node *t = Q->Tail;
[35]     t->n = x;             // @1
[36]     Q->Tail = x;
[37]   unlock(&Q->TLock);
[38] }

[40] data_type dequeue(Queue *Q){
[41]   lock(&Q->HLock);
[42]     Node *h = Q->Head;
[43]     Node *s = h->n;       // @2
[44]     if (s == NULL)
[45]       unlock(&Q->HLock);
[46]       return EMPTY;
[47]     data_type r = s->d;
[48]     Q->Head = s;
[49]   unlock(&Q->HLock);
[50]   return r;
[51] }
```

(b) Concurrent queue procedures

Figure A.2: Two-lock queue: (a) type definitions; (b) implementation; (c) most general client.



```
[10] #define EMPTY -1
[11] #define TRUE 1
[12] #define FALSE 0
[13] typedef int data_type;
[14] typedef struct node_t {
[15]   data_type d;
[16]   struct node_t *n;
[17]   lock_type NLock;
[18] } Node;
[19] typedef struct set_t {
[20]   struct node_t *Head;
[21] } Set;
```

(a) `Set` and `Node` type definitions

```
[90] void client(Set *s) {
[91]   do {
[92]     if (?)
[93]       add(s, rand());
[94]     else
[95]       remove(s, rand());
[96]   } while (1);
[97] }
```

(c) The most general client of `Set`

```
[100] void initialize(Set *S) {
[101]   Node *Hdummy = alloc(sizeof(Node));
[102]   Hdummy->d = MIN_VALUE;
[103]   Node *Tdummy = alloc(sizeof(Node));
[104]   Tdummy->d = MAX_VALUE;
[105]   Hdummy->n = Tdummy;
[106]   S->Head = Hdummy;
[107] }
```

(d) `Initialize` procedure

```
[30] boolean add(Set *S, data_type v){
[31]   Node *pred = S->Head;
[32]   lock(&pred->NLock);
[33]   Node *curr = pred->n;
[34]   lock(&curr->NLock);
[35]   while (curr->d < v) {
[36]     unlock(&pred->NLock);
[37]     pred = curr;
[38]     curr = curr->n;
[39]     lock(&curr->NLock);
[40]   };
[41]   if (curr->d ≠ v)
[42]     Node *x = alloc(sizeof(Node));
[43]     x->d = v;
[44]     x->n = curr;
[45]     pred->n = x;
[46]     boolean r = TRUE;
[47]   else
[48]     boolean r = FALSE;
[49]   unlock(&pred->NLock);      // @1
[50]   unlock(&curr->NLock);
[51]   return r;
[52] }

[60] boolean remove(Set *S, data_type v){
[61]   Node *pred = S->Head;
[62]   lock(&pred->NLock);
[63]   Node *curr = pred->n;
[64]   lock(&curr->NLock);
[65]   while (curr->d < v) {
[66]     unlock(&pred->NLock);
[67]     pred = curr;
[68]     curr = curr->n;
[69]     lock(&curr->NLock);
[70]   };
[71]   if (curr->d == v)
[72]     Node *s = curr->n;
[73]     pred->n = s;
[74]     boolean r = TRUE;
[75]   else
[76]     boolean r = FALSE;
[77]   unlock(&pred->NLock);      // @2
[78]   unlock(&curr->NLock);
[79]   return r;
[80] }
```

(b) `add` and `remove` procedures

Figure A.3: Pessimistic set: (a) type definitions; (b) implementation of `add` and `remove`; (c) most general client; (d) implementation of `initialize`.

	Data Structure	Mutation	Time	# States
1	Stack [Tre86]	(a)	4.646	618
		(b)	1.407	170
2	Nonblocking queue [MS96]	(a)	202.929	18611
		(b)	8.531	1021
3	Two-lock queue [MS96]	(a)	4.104	766
		(b)	3.667	435
4	Pessimistic set [VHHS06]	(a)	118.467	2981

Table A.1: Results of mutation experiments. 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.

A.2 Mutation Experiments

Tab. A.1 summarizes the mutation experiments (described below). All the experiments were performed with two threads, each executing the data structure's most general client procedure. In all of the experiments, the analysis reported that the data structure may not be linearizable.

Note that none of the code mutations we used is observable in a *sequential* setting. We verified that by running each of the mutation experiments with a single thread executing the most general client procedure.

Stack

- Mutation (a): Line `[26]` of the `push` procedure was replaced by the following non-atomic code fragment:

  ```
  [26a] } while (S->Top ≠ t);
  [26b] S->Top = x;
  ```

  This mutation essentially breaks the atomic `CAS(&S->Top,t,x)` statement of line `[26]` into two separate statements: one that compares the value of `S->Top` with the value of `t`, and another that stores the value of `x` in `S->Top`. Line `[26b]` was specified as the linearization point.

- Mutation (b): Analogous mutation to line `[36]` of the `pop` procedure.

Nonblocking Queue

- Mutation (a): Lines `[27]`, `[30]` and `[31]` of the `enqueue` procedure were omitted. This means that the procedure attempts to append the new node to the tail of the queue without checking that the tail points to the last node in the list (and without trying to advance it in case it doesn't).

- Mutation (b): Lines `[47]` and `[49]` of the `dequeue` procedure were omitted. This means that the procedure returns `EMPTY` if the tail of the queue points to the head dummy, without checking that the dummy is the last node in the list (and without trying to advance the tail in case it isn't).

(In both experiments, the specification of the linearization points was not changed.)

Two-Lock Queue

- Mutation (a): The code of the `enqueue` procedure was mutated by adding the following code fragment between lines `[35]` and `[36]`:

  ```
  [35a] unlock(&Q->TLock);
  [35b] lock(&Q->TLock);
  ```

  This mutation breaks the continuity of the executing thread's ownership of the tail-lock.

- Mutation (b): The code of the `dequeue` procedure was mutated in a similar way by omitting line `[45]` and adding the following code lines after line `[43]` and after line `[47]`, respectively:

  ```
  [43a] unlock(&Q->HLock);
  ```

```
[47a] lock(&Q->HLock);
```

(In both experiments, the specification of the linearization points was not changed.)

Pessimistic Set

- Mutation (a): Lines `[50]` and `[78]` were wrongly specified as the linearization points of procedures `add` and `remove`, respectively.

  To see why these linearization points are incorrect, consider, for example, the following scenario: Threads A and B concurrently try to add the same value v to the set. Thread A starts executing, successfully adds v to the set and unlocks `pred`. At this point, thread B starts executing. Since the new node added by A is accessible to B, B finds that the value v is already in the set, and returns `FALSE`. Finally, A resumes execution, unlocks `curr` and returns `TRUE`.

  If we specify `unlock(curr)` as the linearization point of the `add` procedure, then in the corresponding atomic execution B's operation is executed before A's, and hence it is thread B that returns `TRUE` while thread A returns `FALSE`.

A.3 Analysis

Operational Specification

In the verification of the stack and queue benchmarks, we used simple sequential implementations as operational specifications. The sequential implementations use the same representation of the data structure as the concurrent implementations (except for the locks of the two-lock queue). The sequential procedures are essentially simplified versions of the concurrent procedures, obtained by omitting unneeded synchronization schemes (e.g.,

locking, retrying, helping and rechecking) and replacing `CAS` statements with regular pointer-update statements.

In the verification of the set benchmark, we used a *declarative* rather than an executable specification. The reference set is represented as a collection of nodes that store its data items and are marked by a special "member" bit. The reference `add` operation simply checks whether the given item appears in the set, and if it doesn't, allocates a new node storing the item and adds it to the collection. The reference `remove` operation checks whether the given item appears in the set, and if it does, removes the node storing it from the collection.

Initialization

Our analysis verifies the linearizability of a concurrent object *with respect to a user-specified initial state*, i.e., it proves the linearizability of every execution that starts in a specified state. In all of our experiments, we begin the analysis with both candidate and reference data structures empty. If the data structures contain dummy nodes in this state, then corresponding dummy nodes are correlated. For example, in the analysis of the queue benchmarks, the candidate head dummy is correlated with the reference head dummy in the initial state.

Utilized Abstractions

Tab. A.2, Tab. A.3, Tab. A.4 and Tab. A.5 summarize the core predicates and the instrumentation predicates used in the analysis of the benchmarks.

Tab. A.2 shows the core predicates used for the stack benchmark and for all the queue benchmarks.

Tab. A.3 shows the instrumentation predicates used to refine the abstraction in the verification of the non-blocking queue benchmark. For the nonblocking queue of [MS96] we recorded for every thread t the successor of the node pointed-to by its local copy of `Tail` (predicate $tSuccessor_t(o)$).

	Predicates	Intended Meaning
1	$f(o)$	field f of the candidate concurrent object points to object o
2	$x_t(o)$	local variable x of candidate thread t points to object o
3	$n(o_1, o_2)$	the candidate `n`-field of object o_1 points to object o_2
4	$uncorrelated(o)$	object o is an uncorrelated candidate object
5	$nonuniform(o)$	object o is a nonuniform duo-object

Table A.2: Core predicates for the stack and queue benchmarks. Predicates (1)-(3) have r-superscripted analogs pertaining to the reference memory state. For example, $f^r(o)$ records that field f of the reference concurrent object points to object o.

Predicates	Defining Formula
$xSuccessor_t(o)$	$\exists(o_1) : x_t(o_1) \wedge n(o_1, o)$

Table A.3: Instrumentation predicates for the nonblocking queue benchmark.

For the optimized version of [DGLM04] we also recorded for every thread t the successor of the node pointed-to by its local copy of `Head` (predicate $hSuccessor_t(o)$).

Tab. A.4 and Tab. A.5 show the core predicates and the instrumentation predicates used for the set benchmark, respectively.

	Predicates	Intended Meaning
1	$f(o)$	field f of the candidate concurrent object points to object o
2	$x_t(o)$	local variable x of candidate thread t points to object o
3	$n(o_1, o_2)$	the candidate `n`-field of object o_1 points to object o_2
4	$uncorrelated(o)$	object o is an uncorrelated candidate object
5	$Tail(o)$	object o is the candidate tail dummy-node
6	$lockedBy[t](o)$	the lock of object o is held by candidate thread t
7	$dle(o_1, o_2)$	the data item of object o_1 is less or equal to the data item of object o_2
8	$member(o)$	object o belongs to the reference set

Table A.4: Core predicates for the set benchmark.

Predicates	Defining Formula	Intended Meaning
$inOrder(o)$	$\forall(o_1) : n(o, o_1) \rightarrow (dle(o, o_1) \wedge \neg dle(o_1, o))$	the data item of object o is less than the data item of o's `n`-successor
$bn(o_1, o_2)$	$n^*(o_1, o_2)$	object o_2 is reachable from object o_1 via a path of `n`-fields
$rt[f](o)$	$\exists(o_1) : f(o_1) \wedge bn(o_1, o)$	object o is reachable from the object pointed-to by field `f` of the candidate concurrent object via a path of `n`-fields
$rt[x_t](o)$	$\exists(o_1) : x_t(o_1) \wedge bn(o_1, o)$	object o is reachable from the object pointed-to by local variable `x` of candidate thread `t` via a path of `n`-fields
$is(o)$	$\exists(o_1, o_2) : o_1 \neq o_2 \wedge n(o_1, o) \wedge n(o_2, o)$	object o is pointed-to by the `n`-fields of two different objects

Table A.5: Instrumentation predicates for the set benchmark. (n^* denotes the reflexive transitive closure on binary predicate n.)

Appendix B

Correlated Memory States

In this section, we formalize the representation of correlated memory states. A *correlated memory state* is essentially a combination of a pair of *concurrent memory states* and some additional information used to maintain the correlation relation and to compare operation results. For simplicity, our semantics allows only for a single concurrent object.

Concurrent Memory States

Fig. B.1(a) defines the semantic domains of *concurrent memory states*, and the meta-variables ranging over them. We assume $l \in \mathit{Loc}$ to be an unbounded set of locations. A value $v \in \mathit{Val}$ is either a location, NULL, or an integer. $t \in \mathcal{T}$ is the domain of thread identifiers. We also assume the syntactic domains $x \in \mathcal{V}$ of variable identifiers, $f \in \mathcal{F}$ of field identifiers, and $pc \in \mathcal{PC}$ of program points.

A *concurrent memory state* $\sigma_S = \langle \mathit{tlsf}, g \rangle \in \Sigma_S$ is a pair: tlsf is a map that associates the identifier of every thread used by the program with its *local state*, and g is the program's *global state*. The *thread-local state* $\mathit{tls} \in \mathcal{TLS}$ of a thread $t \in \mathcal{T}$ contains t's program counter $pc \in \mathcal{PC}$ and an environment $\rho \in \mathcal{E}$, mapping t's local variables to their current values. The local state $\mathit{tlsf}(t)$ is accessible only to t. The *global state* $g = \langle A, h, o \rangle \in \mathcal{G}$ consists

$$\begin{array}{lcl} \rho & \in & \mathcal{E} = \mathcal{V} \hookrightarrow \mathit{Val} \\ \mathit{tls} & \in & \mathcal{TLS} = \mathcal{PC} \times \mathcal{E} \\ \mathit{tlsf} & \in & \mathit{TLSF} = \mathcal{T} \hookrightarrow \mathcal{TLS} \\ h & \in & \mathcal{H} = \mathit{Loc} \hookrightarrow (\mathcal{F} \hookrightarrow \mathit{Val}) \\ g & \in & \mathcal{G} = 2^{\mathit{Loc}} \times \mathcal{H} \times (\mathcal{F} \hookrightarrow \mathit{Val}) \\ \sigma_S & \in & \Sigma_S = \mathit{TLSF} \times \mathcal{G} \end{array}$$

(a) standard semantics

$$\begin{array}{lcl} \mathit{ret} & \in & \mathcal{R} = \mathcal{T} \hookrightarrow \mathit{Val} \\ \mathit{la} & \in & \mathcal{LA} = \mathcal{T} \hookrightarrow \mathit{Loc} \\ \phi & \in & \Phi = \mathit{Loc} \hookrightarrow \mathit{Loc} \\ \sigma_C & \in & \Sigma_C = \Sigma_S \times \Sigma_S \times \Phi \times \mathcal{LA} \times \mathcal{R} \end{array}$$

(b) correlating semantics

$$\begin{array}{lcl} \delta_L & \in & \Delta_L = 2^{\mathit{Loc}} \times 2^{\mathit{Loc}} \times 2^{\mathit{Loc}} \\ \delta_M & \in & \Delta_M = \Delta_L \times \mathcal{H} \times (\mathcal{F} \hookrightarrow \mathit{Val}) \\ \delta_S & \in & \Delta_S = \mathit{TLSF} \times \Delta_M \\ \sigma_\Delta & \in & \Sigma_\Delta = \Sigma_S \times \Delta_S \times \mathcal{LA} \times \mathcal{R} \end{array}$$

(c) delta semantics

Figure B.1: Semantic domains of the standard, the correlating, and the delta semantics.

of a set $A \subset 2^{\mathit{Loc}}$ of allocated memory locations and a global heap $h \in \mathcal{H}$, mapping fields of allocated objects to their current values. The concurrent object is represented by a mapping $o \in \mathcal{F} \hookrightarrow \mathit{Val}$ of its fields to their values.

Correlated Memory States

Fig. B.1(b) defines the semantic domains of correlated memory states. A *correlated memory state* $\sigma_C = \langle \sigma_S^c, \sigma_S^r, \phi, \mathit{la}, \mathit{ret} \rangle \in \Sigma_C$ is a 5-tuple: The concurrent memory states $\sigma_S^c = \langle \mathit{tlsf}^c, \langle A^c, h^c, o^c \rangle \rangle \in \Sigma_S$ and $\sigma_S^r = \langle \mathit{tlsf}^r, \langle A^r, h^r, o^r \rangle \rangle \in \Sigma_S$ are the candidate and the reference components, respectively. They both have the same (finite) set of threads, $\mathrm{dom}(\mathit{tlsf}^c) = \mathrm{dom}(\mathit{tlsf}^r)$. The correlation relation is an injective partial function, $\phi \in A^c \hookrightarrow A^r$, mapping candidate objects to their correlated reference objects.

The $la \in \mathcal{T} \hookrightarrow Loc$ map assists in constructing the correlation function by recording the last location allocated by every candidate thread during its current operation. When a thread t executing a reference operation allocates an object l^r, the semantics updates the correlation function $\phi' = \phi[la(t) \mapsto l^r]$. (In all our benchmarks, every candidate operation and every reference operation allocates, at most, a single object. Furthermore, the candidate operation allocates before the corresponding reference operation. Thus, la suffices to maintain the correlation. If an operation may allocate several objects, or if the reference allocation may precede the candidate allocation, more complicated machinery might be required.)

The $ret \in \mathcal{T} \hookrightarrow Val$ map saves the result of the last reference operation for every thread. We assume that every procedure *proc* always writes its return value to a designated thread-local variable r of the invoking thread. Furthermore, this variable is expected to get its value from the data field of an allocated node, or be set to some predefined constant value (either a boolean value or a special error code). The *ret* map records for every thread the location of the node in the former case, or the constant value in the latter. When a thread t's candidate procedure *proc* returns, the semantics compares *proc*'s return value against $ret(t)$ and checks that the return value is either a constant value equal to $ret(t)$ or is taken from a node correlated with the node pointed-to by $ret(t)$.

Appendix C

Correlated Operational Semantics

In this section, we formalize the correlated operational semantics.

C.1 Standard Semantics for Multi-Threaded Programs

Program Syntax

We consider programs written in a language with the set of primitive statements shown in Tab. C.1.

We use a control-flow graph *(CFG)* to represent a sequential program executed by a thread $t \in \mathcal{T}$. The control-flow graph consists of a set of vertices ($N_t \subset \mathcal{PC}$), a set of edges ($E_t \subseteq N_t \times N_t$), a designated entry vertex (n_t), and a map (M_t) that associates every edge with a primitive statement.

In addition to the statements in Tab. C.1, we define two special statements for procedure invocation and response (see Tab. C.3). We assume the syntactic domain $proc \in Proc$ of procedure identifiers. For procedure $proc \in Proc$, we denote by m_{proc} the number of formal parameters of $proc$

and by $\{z_i^{proc} | 1 \leq i \leq m_{proc}\}$ the sequence of *proc*'s formal parameters. A procedure invocation initializes the formal parameters to the values of the actual parameters. A procedure's response writes the return value to a designated thread-local variable r. We assume that the assignment to r is either of the form $r = c$ for some constant c or of the form $r = y \rightarrow d$ where y is a local variable and d is the data field of a node object.

Program Semantics

Programs in our language are executed using a standard two-level store semantics for pointer programs (see, e.g., [MS77, Rey02]). We define the *meaning* of statements using an auxiliary semantic domain of *thread-observable states*, $\sigma_T \in \Sigma_T = \mathcal{E} \times \mathcal{G}$. A thread-observable state, $\sigma_T \in \Sigma_T$, of thread $t \in \mathcal{T}$ consists of t's local environment, $\rho \in \mathcal{E}$, and the global state of the multi-threaded program, $g \in \mathcal{G}$. The meaning of every statement st is given as a binary relation $[\![st]\!] \subseteq \Sigma_T \times \Sigma_T$, with the intention that $\langle \sigma_T, \sigma'_T \rangle \in [\![st]\!]$ iff the execution of st in memory state σ_T may lead to memory state σ'_T. Tab. C.2 specifies the meaning of the primitive statements shown in Tab. C.1. Tab. C.3 specifies the meaning of invocation and response statements. Note that for a response statement of the form $res(proc, y \rightarrow d)$, we don't actually track the integer data value of the object pointed-to by y, but the object's location. We use $\langle st, \sigma_T \rangle \rightsquigarrow \sigma'_T$ as an equivalent notation for $\langle \sigma_T, \sigma'_T \rangle \in [\![st]\!]$.

The behavior of a thread $t \in \mathcal{T}$ in a multi-threaded program can be described by a transition relation, $tr_t \subseteq (\mathcal{TLS} \times \mathcal{G}) \times (\mathcal{TLS} \times \mathcal{G})$, satisfying

$$\langle \langle \langle pc, \rho \rangle, g \rangle, \langle \langle pc', \rho' \rangle, g' \rangle \rangle \in tr_t \text{ iff } \langle \langle \rho, g \rangle, \langle \rho', g' \rangle \rangle \in [\![M_t(\langle pc, pc' \rangle)]\!].$$

The behavior of a multi-threaded program can be described by a transition relation, $tr_S \subseteq \Sigma_S \times \Sigma_S$, that interleaves the execution of different threads. I.e.,

$$\langle \sigma_S, \sigma'_S \rangle \in tr_S \text{ iff } \exists t \in \mathcal{T} : \sigma_S \overset{t}{\rightsquigarrow} \sigma'_S$$

Statement	Intended meaning
`nop`	A no-operation statement
$x = a$	Assign the value of constant a to variable x
$x = y$	Copy the value of variable y to variable x
$x = y \rightarrow f$	Copy the value of the f-field of the object pointed-to by variable y to variable x
$x \rightarrow f = y$	Copy the value of variable y to the f-field of the object pointed-to by variable x
$x = alloc()$	Allocate a fresh object and assign its address to variable x
$x = read(f)$	Assign to variable x the value of the f-field of the concurrent object
$CAS(\&(f), y_1, y_2)$	CAS operation on the f-field of the concurrent object
$CAS(\&(x \rightarrow f), y_1, y_2)$	CAS operation on the f-field of the object pointed-to by variable x
$assume(x \bowtie y)$	There is a $\bowtie$ relation between the values of variables x and y
$assume(x \bowtie a)$	There is a $\bowtie$ relation between the values of variable x and constant a
$assume(x \bowtie read(f))$	There is a $\bowtie$ relation between the value of variable x and the value of the f-field of the concurrent object
$assume((x \rightarrow f) \diamond y)$	There is a $\bowtie$ relation between the value of variable y and the value of the f-field of the object pointed-to by variable x

Table C.1: The set of primitive statements. *assume* statements handle conditions. ($\bowtie$ stands for either $=$ or $\neq$. $\diamond$ stands for a numeric relation ($=$, $\neq$, $<$, $>$, $\leq$ or $\geq$)).

Transition	Side Condition
$\langle \texttt{nop}, \sigma_T \rangle \rightsquigarrow \sigma_T$	
$\langle x = a, \sigma_T \rangle \rightsquigarrow \langle \rho[x \mapsto [\![a]\!]], \langle A, h, o \rangle \rangle$	
$\langle x = y, \sigma_T \rangle \rightsquigarrow \langle \rho[x \mapsto \rho(y)], \langle A, h, o \rangle \rangle$	
$\langle x = y \rightarrow f, \sigma_T \rangle \rightsquigarrow \langle \rho[x \mapsto h(\rho(y))f], \langle A, h, o \rangle \rangle$	$\rho(y) \neq NULL$
$\langle x \rightarrow f = y, \sigma_T \rangle \rightsquigarrow \langle \rho, \langle A, h[(\rho(x), f) \mapsto \rho(y)], o \rangle \rangle$	$\rho(x) \neq NULL$
$\langle x = alloc(), \sigma_T \rangle \rightsquigarrow \langle \rho[x \mapsto l], \langle A \cup \{l\}, h \cup I(l), o \rangle \rangle$	$l \notin A$
$\langle x = read(f), \sigma_T \rangle \rightsquigarrow \langle \rho[x \mapsto o(f)], \langle A, h, o \rangle \rangle$	
$\langle CAS(\&(f), y_1, y_2), \sigma_T \rangle \rightsquigarrow \langle \rho, \langle A, h, o[f \mapsto \rho(y_2)] \rangle \rangle$	$o(f) = \rho(y_1)$
$\langle CAS(\&(x \rightarrow f), y_1, y_2), \sigma_T \rangle \rightsquigarrow \langle \rho, \langle A, h[(\rho(x), f) \mapsto \rho(y_2)], o \rangle \rangle$	$\rho(x) \neq NULL,\ h(\rho(x))f = \rho(y_1)$
$\langle assume(x \bowtie y), \sigma_T \rangle \rightsquigarrow \sigma_T$	$\rho(x) \bowtie \rho(y)$
$\langle assume(x \bowtie a), \sigma_T \rangle \rightsquigarrow \sigma_T$	$\rho(x) \bowtie [\![a]\!]$
$\langle assume(x \bowtie read(f)), \sigma_T \rangle \rightsquigarrow \sigma_T$	$\rho(x) \bowtie o(f)$
$\langle assume((x \rightarrow f) \diamond y), \sigma_T \rangle \rightsquigarrow \sigma_T$	$h(\rho(x))f \diamond \rho(y)$

Table C.2: Meaning of statements. $\sigma_T = \langle \rho, \langle A, h, o \rangle \rangle$. $[\![a]\!]$ denotes a's semantic value. $I(l)$ initializes all pointer fields at l to $NULL$, all integer fields to 0 and all boolean fields to $false$. The side-conditions ensure that the program does not dereference a null-valued pointer: The execution of the program halts if the dereferenced variable has a $NULL$ value. An allocated location is guaranteed to be *fresh*, i.e., it is not used in the current memory state.

$\langle inv(proc, x_1, \ldots, x_{m_{proc}}), \sigma_T \rangle \rightsquigarrow \langle \rho[z_i^{proc} \mapsto \rho(x_i),\ 1 \leq i \leq m_{proc}], \langle A, h, o \rangle \rangle$
$\langle res(proc, c), \sigma_T \rangle \rightsquigarrow \langle \rho[r \mapsto [\![c]\!]], \langle A, h, o \rangle \rangle$
$\langle res(proc, y \rightarrow d), \sigma_T \rangle \rightsquigarrow \langle \rho[r \mapsto \rho(y)], \langle A, h, o \rangle \rangle$

Table C.3: Meaning of procedure invocation (inv) and response (res) statements. $\sigma_T = \langle \rho, \langle A, h, o \rangle \rangle$.

where $\overset{t}{\rightsquigarrow}$ is an auxiliary relation that describes the change in program state as a result of the execution of a single statement by thread $t \in \mathcal{T}$:

$$\langle tlsf, g\rangle \overset{t}{\rightsquigarrow} \langle tlsf[t \mapsto tls'], g'\rangle \text{ iff } \langle\langle tlsf(t), g\rangle, \langle tls', g'\rangle\rangle \in tr_t.$$

C.2 Correlating Semantics

Main Idea

The correlating semantics alternates between execution in the candidate memory state and execution in the reference memory state. The additional information maintained by the correlating semantics (i.e., ϕ, la, and ret) has no effect on either execution, except that the correlating semantics aborts if the return value of a candidate operation does not match the return value of its corresponding reference operation.

The alternation between the candidate and the reference execution is done using control structures similar to those of [Fla04]. Fig. C.1 shows the utilized auxiliary functions.

The functions $isOutside^r$ and $phase^c$ map every thread to its *execution phase* in the reference and candidate state, respectively. For thread $t \in \mathcal{T}$ and reference memory state $\sigma_S^r \in \Sigma_S$, $isOutside^r(t, \sigma_S^r)$ holds iff t is *outside* any operation in σ_S^r (this can be determined by t's program counter in σ_S^r). For thread $t \in \mathcal{T}$ and candidate memory state $\sigma_S^c \in \Sigma_S$, $phase^c(t, \sigma_S^c) = Outside$ if t is *outside* any operation in σ_S^c, $phase^c(t, \sigma_S^c) = PreLin$ if t is *inside* an operation *before* reaching the linearization point, and $phase^c(t, \sigma_S^c) = PostLin$ if t is *inside* an operation *after* passing the linearization point. We assume, for simplicity, that this information can be determined by t's program counter in σ_S^c. (This assumption does not hold for the nonblocking queue benchmark. See App. A.1 for the treatment of this case.)

Note that $phase^c$ represents the user's choice of fixed linearization points. A candidate thread moves from the $Outside$ phase to the $PreLin$ phase when

$isOutside^r : \mathcal{T} \times \Sigma_S \to \{true, false\}$
$phase^c : \mathcal{T} \times \Sigma_S \to \{Outside, PreLin, PostLin\}$
$start^r : Proc \to \mathcal{PC}$
$getProc : \mathcal{PC} \to Proc$

Figure C.1: Auxiliary functions for the correlating semantics.

it executes a procedure invocation statement. It remains in the $PreLin$ phase until it executes the statement associated with the CFG edge specified as the procedure's linearization point, at which time it moves to the $PostLin$ phase. It then remains in the $PostLin$ phase until it executes the procedure's response statement and returns to the $Outside$ phase.

By keeping track of threads' execution phases, the correlating semantics is able to trigger the reference operation at the linearization point of the candidate operation, and guarantee that reference operations are executed atomically. Note that the values of the arguments for the invocation of the reference operation can be obtained from the candidate memory state. (Without loss of generality, we assume formal parameters are not modified.)

If the correlating semantics detects a linearizability violation (i.e., the return value of a candidate operation does not match the return value of its corresponding reference operation), it derives a special *wrong* state. For simplicity, the correlating semantics gets stuck if it encounters a runtime error in the candidate execution. (Alternatively, we could define a special *error* state that is derived in this case.) Our analysis reports the detection of both linearizability violations and runtime errors.

Transition Rules

Fig. C.2 and Fig. C.3 define the transition rules of the correlating semantics. We denote the correlating semantics' transition relation by $\Rightarrow$.

We use a separate control-flow graph *(CFG)* to represent the sequential implementation of each procedure $proc \in Proc$. The function $start^r$ (shown

in Fig. C.1) maps every procedure to the initial program point of its sequential implementation. We denote by M^r the map that associates every edge in the control flow graphs of the sequential implementations with a primitive statement. We denote by $\overset{proc,t}{\rightsquigarrow}$ the transition relation that describes the change in the reference memory state as a result of the execution of a single statement of procedure *proc* by thread $t \in \mathcal{T}$.

The function $getProc$ (shown in Fig. C.1) maps a program point of the candidate implementation to the procedure to which it belongs. (This information is used to determine which reference procedure should be triggered at the linearization point.)

$\sigma_S^c = \langle tlsf^c, g^c \rangle$	$\sigma_S^{c\prime} = \langle tlsf^{c\prime}, g^{c\prime} \rangle$
$\sigma_S^r = \langle tlsf^r, g^r \rangle$	$\sigma_S^{r\prime} = \langle tlsf^{r\prime}, g^{r\prime} \rangle$
$tlsf^c(t) = \langle pc_t^c, \rho_t^c \rangle$	$tlsf^{c\prime}(t) = \langle pc_t^{c\prime}, \rho_t^{c\prime} \rangle$
$tlsf^r(t) = \langle pc_t^r, \rho_t^r \rangle$	$tlsf^{r\prime}(t) = \langle pc_t^{r\prime}, \rho_t^{r\prime} \rangle$

(a) Notations

$$\frac{\begin{array}{l}\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r) \\ \sigma_S^c \overset{t}{\rightsquigarrow} \sigma_S^{c\prime} \\ (\ phase^c(t, \sigma_S^c) = Outside \wedge phase^c(t, \sigma_S^{c\prime}) = PreLin, \text{ or} \\ \ \ phase^c(t, \sigma_S^c) = PreLin \wedge phase^c(t, \sigma_S^{c\prime}) = PreLin, \text{ or} \\ \ \ phase^c(t, \sigma_S^c) = PostLin \wedge phase^c(t, \sigma_S^{c\prime}) = PostLin \quad)\end{array}}{\langle \sigma_S^c, \sigma_S^r, \phi, la, ret \rangle \Rightarrow \langle \sigma_S^{c\prime}, \sigma_S^r, \phi, la', ret \rangle}$$

where

$$la' = \begin{cases} la[t \mapsto l^c], M_t(\langle pc_t^c, pc_t^{c\prime} \rangle) = alloc() \text{ and } l^c \text{ is the allocated location} \\ la, \text{otherwise} \end{cases}$$

(b) [*candidate*] rule

$$\frac{\begin{array}{l}\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r) \\ \sigma_S^c \overset{t}{\rightsquigarrow} \sigma_S^{c\prime} \\ phase^c(t, \sigma_S^c) = PreLin \wedge phase^c(t, \sigma_S^{c\prime}) = PostLin \\ g^{r\prime} = g^r \\ tlsf^{r\prime} = tlsf^r[t \mapsto \langle pc_t^{r\prime}, \rho_t^{r\prime} \rangle] \\ proc = getProc(pc_t^c) \\ pc_t^{r\prime} = start^r(proc) \\ \rho_t^{r\prime} = \rho_t^r[z_i^{proc} \mapsto \rho_t^c(z_i^{proc}),\ 1 \le i \le m_{proc}]\end{array}}{\langle \sigma_S^c, \sigma_S^r, \phi, la, ret \rangle \Rightarrow \langle \sigma_S^{c\prime}, \sigma_S^{r\prime}, \phi, la', ret \rangle}$$

where

$$la' = \begin{cases} la[t \mapsto l^c], M_t(\langle pc_t^c, pc_t^{c\prime} \rangle) = alloc() \text{ and } l^c \text{ is the allocated location} \\ la, \text{otherwise} \end{cases}$$

(c) [*lin-point*] rule

Figure C.2: Transition rules of the correlating semantics (part I). The transition below the solid line is enabled iff the conditions above the solid line are met.

$$\frac{\neg isOutside^r(t, \sigma_S^r) \quad \sigma_S^r \overset{proc,t}{\rightsquigarrow} \sigma_S^{r\prime}}{\langle \sigma_S^c, \sigma_S^r, \phi, la, ret\rangle \Rightarrow \langle \sigma_S^c, \sigma_S^{r\prime}, \phi', la, ret'\rangle}$$

where

$$\phi' = \begin{cases} \phi[la(t) \mapsto l^r], M^r(\langle pc_t^r, pc_t^{r\prime}\rangle) = alloc() \text{ and } l^r \text{ is the allocated location} \\ \phi, \text{otherwise} \end{cases}$$

$$ret' = \begin{cases} ret[t \mapsto \rho_t^{r\prime}(r)], isOutside^r(t, \sigma_S^{r\prime}) \\ ret, \text{otherwise} \end{cases}$$

$$proc = getProc(pc_t^c)$$

(d) [*reference*] rule.

$$\frac{\begin{array}{l}\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r) \\ \sigma_S^c \overset{t}{\rightsquigarrow} \sigma_S^{c\prime} \\ phase^c(t, \sigma_S^c) = PostLin \wedge phase^c(t, \sigma_S^{c\prime}) = Outside \\ \rho_t^{c\prime}(r) = m_\phi^{-1}(ret(t))\end{array}}{\langle \sigma_S^c, \sigma_S^r, \phi, la, ret\rangle \Rightarrow \langle \sigma_S^{c\prime}, \sigma_S^r, \phi, la, ret\rangle}$$

(e) [*finish*] rule

$$\frac{\begin{array}{l}\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r) \\ \sigma_S^c \overset{t}{\rightsquigarrow} \sigma_S^{c\prime} \\ phase^c(t, \sigma_S^c) = PostLin \wedge phase^c(t, \sigma_S^{c\prime}) = Outside \\ \rho_t^{c\prime}(r) \neq m_\phi^{-1}(ret(t))\end{array}}{\langle \sigma_S^c, \sigma_S^r, \phi, la, ret\rangle \Rightarrow wrong}$$

(f) [*wrong*] rule

Figure C.3: Transition rules of the correlating semantics (part II). (m_ϕ is the map defined in Fig. D.1.)

Appendix D

Delta Abstraction

In this section, we formalize the delta abstraction. We abstract correlated memory states using a 2-step successive abstraction. In the first step, we apply a novel *delta abstraction*, which explicitly represents the candidate memory state, and implicitly represents the reference memory state by recording the differences that distinguish it from the candidate memory state. In the second phase, we bound delta states into *abstract delta memory states* using *canonical abstraction* [SRW02].

Step I: Delta Abstraction

The delta abstraction abstracts away only the *names* of the *locations* of correlated reference objects and conjoins the representation of the candidate and reference states.

Fig. B.1(c) defines the semantic domains of delta memory states. The function *toDelta*, defined in Fig. D.1, maps a correlated memory state to a delta memory state. Function *toDelta* uses two auxiliary functions: m_ϕ maps every correlated candidate location to its corresponding reference location (and acts as the identity function for all other values); $sim_\phi(h^c, h^r)$ is a relation that defines the set of similar locations.

The candidate state is represented as is. Every correlated reference ob-

$$
\begin{array}{l}
\mathit{toDelta}\colon \Sigma_C \to \Sigma_\Delta \text{ s.t.} \\
\mathit{toDelta}(\langle \sigma_S^c, \sigma_S^r, \phi, la, ret\rangle) \stackrel{\text{def}}{=} \langle \sigma_S^c, \delta_S, la, ret'\rangle \\
\text{where} \\
\quad \sigma_S^c \;=\; \langle tlsf^c, \langle A^c, h^c, o^c\rangle\rangle \\
\quad \sigma_S^r \;=\; \langle tlsf^r, \langle A^r, h^r, o^r\rangle\rangle \\
\quad \phi \;\in\; A^c \hookrightarrow A^r \\
\quad \delta_S \;=\; \langle tlsf', \langle\langle \delta_L^c, \delta_L^r, \delta_L^{ns}\rangle, \delta_h, o'\rangle\rangle \\
\qquad tlsf'(t) \;=\; \langle pc, m_\phi^{-1} \circ \rho\rangle \text{ where } tlsf^r(t) = \langle pc, \rho\rangle \\
\qquad o' \;=\; m_\phi^{-1} \circ o^r \\
\qquad \delta_L^c \;=\; A^c \setminus dom(\phi) \\
\qquad \delta_L^r \;=\; A^r \setminus img(\phi) \\
\qquad \delta_L^{ns} \;=\; A^c \setminus dom(sim_\phi(h^c, h^r)) \\
\qquad \delta_h \;=\; m_\phi^{-1} \circ h^r|_{A^r \setminus img(sim_\phi(h^c,h^r))} \circ m_\phi \\
\quad ret' \;=\; m_\phi^{-1} \circ ret \\
\quad m_\phi(v) \;=\; \begin{cases} \phi(v) & v \in dom(\phi) \\ v & \text{otherwise} \end{cases} \\
\quad (l^c, l^r) \in sim_\phi(h^c, h^r) \iff l^r = \phi(l^c) \wedge \\
\qquad\qquad\qquad\qquad \forall f \in \mathcal{F}\colon h^c(l^c)f = m_\phi^{-1}(h^r(l^r)f)
\end{array}
$$

Figure D.1: The function *toDelta* mapping correlated states to delta states

ject l is represented by its correlated candidate counterpart $\phi^{-1}(l)$, which we refer to as a *duo-object*. The sets δ_L^c and δ_L^r record the non-correlated candidate and reference locations, respectively. The set δ_L^{ns} records the non-similar correlated candidate locations, which represent *nonuniform duo-objects*. ($dom(\phi) \setminus \delta_L^{ns}$ contains the *uniform duo-objects*.) The function δ_h records the values of fields for non-correlated reference locations and for non-similar correlated reference locations. The functions ret' and o' redirect reference return values and fields of the reference concurrent object to locations representing them in the delta state. Similarly, the function $tlsf'$ redirects the values of reference local variables to locations representing them in the delta state.

Note that there is no loss of information when converting a correlated

state into a delta state, besides the location names, which are unobservable. We assume, without loss of generality, that the location sets A^c and A^r are disjoint. (This can be guaranteed by applying renaming prior to the delta abstraction).

Step II: Bounded Delta Abstraction

We bound delta memory states by converting them into logical structures and employing canonical abstraction [SRW02].

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of 1/2 (unknown) for cases in which predicates could have either value, 1 (true) or 0 (false). The information partial order on the set $\{0, 1/2, 1\}$ is defined as $0 \sqsubseteq 1/2 \sqsupseteq 1$, and $0 \sqcup 1 = 1/2$.

A 3-*valued logical structure* $S{=}\langle U^S, \iota^S\rangle$ is a pair where U^S is the universe of the structure and ι^S is an interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p)\colon U^{S^k} \to \{0, 1/2, 1\}$. A 2-valued structure is a 3-valued structure with an interpretation limited to $\{0, 1\}$. The set of *2-valued* logical structures is denoted by *2-Struct*, and the set of *3-valued* logical structures is denoted by *3-Struct*.

We convert a delta memory state,

$$\sigma_\Delta = \langle\langle tlsf^c, \langle A^c, h^c, o^c\rangle\rangle, \langle tlsf', \langle\langle \delta_L^c, \delta_L^r, \delta_L^{ns}\rangle, \delta_h, o'\rangle\rangle, la, ret'\rangle,$$

into a 2-valued logical structure $S = \langle U^S, \iota^S\rangle$ in the following manner: We define S's universe to be the union of all locations in the state, i.e., $U^S = A^c \cup \delta_L^r$ (recall that A^c and A^r are disjoint). We use unary predicates to record the values of variables (i.e., the environment components of $tlsf^c$ and $tlsf'$); fields of the concurrent object (i.e., o^c and o'); the last object allocated by every candidate thread (i.e., la); and the reference operations' saved return values (i.e., ret). We use binary predicates to record the values of pointer-

fields of heap-allocated objects, i.e., fields that hold either the location of another heap-allocated object or a *NULL* value (kept in h^c and δ_h). We also use unary predicates to record which location set an object belongs to (δ_L^c, δ_L^r, δ_L^{ns} or $dom(\phi) \setminus \delta_L^{ns}$). We use a designated binary predicate *eq* to track *identity equality in the delta-state*.

The interpretation of the above predicates is taken directly from the delta state. In essence, the resulting 2-valued structure contains all the information which is relevant for the program, except for numerical values.

We establish a Galois connection between the set of program states (ordered by set inclusion) and *3-Struct* using a *canonical abstraction* as a *representation function* mapping a 2-valued state to its "most-precise representation" in *3-Struct* (e.g., see [NNH99]).

A *3-valued* logical structure $S^\sharp$ is a **canonical abstraction** of a *2-valued* logical structure S if there exists a surjective function $f\colon U^S \to U^{S^\sharp}$ satisfying the following conditions: (i) For all $u_1, u_2 \in U^S$, $f(u_1) = f(u_2)$ iff for all unary predicates $p \in \mathcal{P}$, $\iota^S(p)(u_1) = \iota^S(p)(u_2)$, and (ii) For all predicates $p \in \mathcal{P}$ of arity k and for all k-tuples $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

$$\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \bigsqcup_{\substack{u_1,\ldots,u_k \in U^s \\ f(u_i)=u_i^\sharp}} \iota^S(p)(u_1, u_2, \ldots, u_k).$$

The set of concrete memory states such that $S^\sharp$ is their canonical abstraction is denoted by $\gamma(S^\sharp)$. Note that *only* for a summary node u, $\iota^{S^\sharp}(eq)(u,u) = 1/2$.

Appendix E

Soundness

In this section, we prove the correctness of the correlating semantics. Along with the equivalence of the concrete delta semantics to the correlating semantics and the soundness of [SRW02]'s framework for program analysis, this implies the soundness of the analysis.

E.1 Preliminary Definitions

Linearizability of a Concurrent Object

Recall that we consider multi-threaded programs in which a collection of sequential threads of control communicate through a shared data structure called a **concurrent object**. A concurrent object provides a finite set of operations that are the only means to manipulate the object. Every operation has a sequence of arguments with which it is invoked, and when it terminates it returns a response that includes its termination condition (normal or exceptional) and its result (return value). (In our formulation, we assume, for simplicity, that an exceptional termination condition is signaled by a special return value). Each thread applies a sequence of operations to the concurrent object, alternately issuing an invocation and receiving the associated response.

The following notations and definitions are based on [HW90]:

- An **invocation event** is denoted by $\langle op(arg^*)\ t\rangle$, where op is an operation name, arg^* is a sequence of argument values, and $t \in \mathcal{T}$ is the invoking thread.
- A **response event** is denoted by $\langle term(res^*)\ t\rangle$ where $term$ is a termination condition and res^* is a sequence of results.
- A **history** is a finite sequence of invocation and response events.
- A history H is **sequential** if the first event of H is an invocation, and each invocation, except possibly the last, is followed by a *matching response* (i.e., a response associated with the same thread).
- A **sequential specification** for an object is a prefix-closed set of sequential histories for that object. A sequential history is **legal** iff it belongs to this set.
- An **operation**, e, in a history is a pair consisting of an invocation, $inv(e)$, and the next matching response, $res(e)$.
- A history H induces an irreflexive **partial order** $<_H$ on operations: $e_0 <_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in H.
- Two histories H_1 and H_2 are **equivalent** if for each thread i, $H_1|i = H_2|i$. (Where, for history H and thread i, $H|i$ denotes the subsequence of all events in H associated with thread i. We assume all histories are **well-formed**, i.e., $H|i$ is always sequential.)
- If H is a history, **complete(H)** is the maximal subsequence of H consisting only of invocations and matching responses. I.e., complete(H) is derived from H by omitting all **pending invocations** (invocations not followed by a matching response).

Definition E.1.1 (linearizability of a history) *A history H is **linearizable** if it can be extended (by appending zero or more response events) to some history H' such that complete(H') is equivalent to some legal sequential history S, and $<_H \subseteq <_S$. S is called a **linearization** of H.*

Definition E.1.2 (linearizability of a concurrent object) *A concurrent object is **linearizable** iff all its concurrent histories are linearizable.*

Auxiliary Definitions

The goal of our analysis is to verify the linearizability of a multi-threaded program P, i.e., to prove that every history of P is linearizable. We denote by $(\sigma_S^c)^0 \in \Sigma_S$ P's initial memory state (where the shared data structure is empty and all threads are at their initial program points).

Definition E.1.3 (sequence) *A **sequence** π over a set M is a total function $\pi \in \{i \in \mathcal{N} \mid 1 \leq i \leq n\} \rightarrow M$ for some $n \in \mathcal{N}$. The **length of a sequence** π, denoted by $|\pi|$, is $|dom(\pi)|$.*

Definition E.1.4 (trace) *A **trace** of a multi-threaded program P with initial state $(\sigma_S^c)^0 \in \Sigma_S$ and transition relation $tr_S \subseteq \Sigma_S \times \Sigma_S$ is a sequence π over the set Σ_S of memory states such that*

1. *$\pi(1) = (\sigma_S^c)^0$, and*
2. *for all $1 \leq i \leq |\pi| - 1$, $\langle \pi(i), \pi(i+1) \rangle \in tr_S$.*

For trace π, we denote by $f_\pi \in \{i \in \mathcal{N} \mid 1 \leq i \leq |\pi| - 1\} \rightarrow \mathcal{T}$ the function that maps every index i to the thread that executes the i^{th} step in π. I.e., $f_\pi(i) = t$ for the thread $t \in \mathcal{T}$ such that $\pi(i) \overset{t}{\rightsquigarrow} \pi(i+1)$. (For simplicity, we assume t is unique. In particular, if threads' CFGs contain no

self-loops, then $f_\pi(i)$ is the unique thread whose program counter changes in the transition from $\pi(i)$ to $\pi(i+1)$).

We denote by $stmt(\pi, i)$ the statement executed by thread $t = f_\pi(i)$ in the transition $\pi(i) \overset{t}{\rightsquigarrow} \pi(i+1)$, i.e., $stmt(\pi, i) = M_t(\langle pc, pc' \rangle)$ where $t = f_\pi(i)$, $\pi(i) = \langle tlsf, g \rangle$, $tlsf(t) = \langle pc, \rho \rangle$, $\pi(i+1) = \langle tlsf', g' \rangle$, and $tlsf'(t) = \langle pc', \rho' \rangle$. If pc is a program point inside a procedure $proc \in Proc$, we use the notation $pr(\pi, i) = proc$.

An *operation execution* describes a set of steps that comprise the execution of a single operation by a single thread (Def. E.1.5).

Definition E.1.5 (operation execution) *Let π be a trace of program P. An **operation execution** of a thread $t \in \mathcal{T}$ in π is a nonempty sequence of indices $i_1 < i_2 < \ldots < i_n$ such that*

- *$f_\pi(i_1) = t$ and $stmt(\pi, i_1)$ is an invocation statement.*
- *If there exists $j > i_1$ such that $f_\pi(j) = t$ and $stmt(\pi, j)$ is a response statement, then i_n is the minimum such j. Otherwise, i_n is the maximum j such that $f_\pi(j) = t$.*
  *(The operation execution is said to be **terminating** in the former case, and **non-terminating** in the latter.)*
- *$\{i_l\}_{l=2}^{n-1} = \{j \mid i_1 < j < i_n \wedge f_\pi(j) = t\}$.*

A *valid guess of linearization points* specifies *exactly* one linearization point for every terminating operation execution, and *at most* one linearization point for every non-terminating operation execution (Def. E.1.6).

Definition E.1.6 (valid guess of linearization points) *Let π be a trace of program P. A **valid guess of linearization points** for π is a (possibly empty) sequence of indices $k_1 < k_2 < ... < k_m$ such that*

- *For every terminating operation execution* $i_1 < i_2 < \ldots < i_n$ *in* π, $|\{i_j\}_{j=1}^{n} \cap \{k_j\}_{j=1}^{m}| = 1$.
- *For every non-terminating operation execution* $i_1 < i_2 < \ldots < i_n$ *in* π, $|\{i_j\}_{j=1}^{n} \cap \{k_j\}_{j=1}^{m}| \leq 1$.
- *For every* j $(1 \leq j \leq m)$, $stmt(\pi, k_j)$ *is neither an invocation statement nor a response statement.*

Given a trace π and a valid guess of linearization points for π, we can construct a *serial execution* in which the operations of π are executed *atomically* in the order of occurrence of their linearization points in π (Def. E.1.7 and Def. E.1.8).

We denote by $(\sigma_S^r)^0 \in \Sigma_S$ the initial reference memory state (where the reference data structure is empty and all threads are outside any operation).

Definition E.1.7 (atomic operation execution) *Let* $\sigma_S^r \in \Sigma_S$ *be a reference memory state satisfying* $\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r)$. *The* ***atomic operation execution*** *of procedure* $proc \in Proc$ *with argument values* $\{v_i\}_{i=1}^{m_{proc}} \subseteq Val$ *by thread* $t \in \mathcal{T}$ *from initial state* σ_S^r, *denoted by* $atm(proc(v_1, \ldots, v_{m_{proc}}), t, \sigma_S^r)$, *is a sequence* π *over* Σ_S *such that*

1. $\pi(1) = \sigma_S^r$
2. $\pi(2) = \langle tlsf^r[t \mapsto \langle pc_t^{r\prime}, \rho_t^{r\prime} \rangle], g^r \rangle$ *where*
   - $\sigma_S^r = \langle tlsf^r, g^r \rangle$
   - $tlsf^r(t) = \langle pc_t^r, \rho_t^r \rangle$
   - $pc_t^{r\prime} = start^r(proc)$
   - $\rho_t^{r\prime} = \rho_t^r[z_i^{proc} \mapsto v_i,\ 1 \leq i \leq m_{proc}]$
3. *for all* $2 \leq i \leq |\pi| - 1$, $\pi(i) \overset{proc,t}{\rightsquigarrow} \pi(i+1)$

4. *for* $i = |\pi|$, $isOutside^r(t, \pi(i))$ *holds*

Note: In order for atomic operation executions to be well defined, we need to make the following assumptions regarding reference operations:

- In every reference memory state $\sigma_S^r \in \Sigma_S$ that satisfies $\forall t \in \mathcal{T} : isOutside^r(t, \sigma_S^r)$, it is always possible to invoke any reference operation by any thread.
- Once a reference operation is invoked, its atomic execution is deterministic and fault-free (i.e., free of runtime errors) and terminates within a finite number of steps.

Definition E.1.8 (serial execution) *Let* π *be a trace of program* P*, and* $k_1 < k_2 < ... < k_m$ *a valid guess of linearization points for* π*. The corresponding* ***serial execution****, denoted by* $\boldsymbol{ser(\pi, k_1 < k_2 < ... < k_m)}$*, is a sequence* $\pi^{ser} = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m$ *over* Σ_S *(where* $\cdot$ *is the concatenation operator) such that*

1. $\pi_1 = atm(proc_1(v_1^1, \ldots, v_{m_{proc_1}}^1), t_1, (\sigma_S^r)^0)$ *where*
   - $proc_1 = pr(\pi, k_1)$
   - $t_1 = f_\pi(k_1)$
   - *for* $1 \leq j \leq m_{proc_1}$, $v_j^1 = \rho_{t_1}^c(z_j^{proc_1})$
     where $\pi(k_1) = \langle tlsf^c, g^c \rangle$ *and* $tlsf^c(t_1) = \langle pc_{t_1}^c, \rho_{t_1}^c \rangle$
2. *for all* $1 < i \leq m$, $atm(proc_i(v_1^i, \ldots, v_{m_{proc_i}}^i), t_i, (\sigma_S^r)_i) = (\sigma_S^r)_i \cdot \pi_i$ *where*
   - $(\sigma_S^r)_i = \pi_{i-1}(|\pi_{i-1}|)$
   - $proc_i = pr(\pi, k_i)$
   - $t_i = f_\pi(k_i)$

- *for* $1 \leq j \leq m_{proc_i}$, $v_j^i = \rho_{t_i}^c(z_j^{proc_i})$
  where $\pi(k_i) = \langle tlsf^c, g^c \rangle$ *and* $tlsf^c(t_i) = \langle pc_{t_i}^c, \rho_{t_i}^c \rangle$

In what follows, we write $ser(\pi, k_1 < k_2 < ... < k_m) = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m$ with the intention that $\{\pi_i\}_{i=1}^m$ are as defined in Def. E.1.8.

We denote by $\pi(j, \ldots, l)$ the subsequence $\pi(j), \pi(j+1), ..., \pi(l)$ of π. For trace π over Σ_S and state $\sigma_S \in \Sigma_S$, we denote by $\pi \times \sigma_S$ the sequence π' over $\Sigma_S \times \Sigma_S$ such that $\pi'(i) = (\pi(i), \sigma_S)$ for $1 \leq i \leq |\pi(i)|$. Similarly, we denote by $\sigma_S \times \pi$ the sequence π'' over $\Sigma_S \times \Sigma_S$ such that $\pi''(i) = (\sigma_S, \pi(i))$ for $1 \leq i \leq |\pi(i)|$.

A *combined execution* describes the alternate execution of a trace of program P and its corresponding serial execution (Def. E.1.9).

Definition E.1.9 (combined execution) *Let π be a trace of program P, and $k_1 < k_2 < ... < k_m$ a valid guess of linearization points for π. Let $ser(\pi, k_1 < k_2 < ... < k_m) = \pi_1 \cdot \pi_2 \cdot \ldots \pi_m$. The corresponding* ***combined execution****, denoted by* $\boldsymbol{comb(\pi, k_1 < k_2 < ... < k_m)}$*, is a sequence over $\Sigma_S \times \Sigma_S$, defined by*

$$comb(\pi, k_1 < k_2 < ... < k_m) = \pi_1^{comb} \cdot \pi_2^{comb} \cdot \ldots \cdot \pi_m^{comb} \cdot \pi_{m+1,c}^{comb}$$

where

- $\pi_1^{comb} = \pi_{1,c}^{comb} \cdot \pi_{1,r}^{comb}$ *where*

$$\pi_{1,c}^{comb} = \pi(1, \ldots, k_1) \times \pi_1(1)$$

$$\pi_{1,r}^{comb} = \pi(k_1 + 1) \times \pi_1(2, \ldots, |\pi_1|)$$

- *for all* $2 \leq j \leq m$: $\pi_j^{comb} = \pi_{j,c}^{comb} \cdot \pi_{j,r}^{comb}$ *where*

$$\pi_{j,c}^{comb} = \pi(k_{j-1} + 2, \ldots, k_j) \times \pi_{j-1}(|\pi_{j-1}|)$$

$$\pi^{comb}_{j,r} = \pi(k_j + 1) \times \pi_j$$

- $\pi^{comb}_{m+1,c} = \pi(k_m + 2, \ldots, |\pi|) \times \pi_m(|\pi_m|)$

E.2 Linearizability of a Program Trace

Definition E.2.1 (witness for linearizability) *Let π be a trace of program P, and $k_1 < k_2 < ... < k_m$ a valid guess of linearization points for π. Then $ser(\pi, k_1 < k_2 < ... < k_m) = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m$ is a* ***witness for the linearizability of π by $k_1 < k_2 < ... < k_m$*** *if for every terminating operation execution $i_1 < i_2 < \ldots < i_n$ in π, the following condition holds:*

Let j be the unique index such that $k_j \in \{i_l\}_{l=1}^{n}$. Let $t = f_\pi(k_j)$ and $proc = pr(\pi, k_j)$. Then the response statements executed by thread t in the transitions $\pi(i_n) \overset{t}{\rightsquigarrow} \pi(i_n + 1)$ and $\pi_j(|\pi_j| - 1) \overset{proc,t}{\rightsquigarrow} \pi_j(|\pi_j|)$ have equal return values.

Note: Response statements have two possible forms (see Tab. C.3). The return value of a response statement of the form "$res(proc, c)$" is $[\![c]\!]$. The return value of a response statement of the form "$res(proc, y \rightarrow d)$" executed in state $\sigma_T = \langle \rho, \langle A, h, o \rangle \rangle$ is $h(\rho(y))d$.

Definition E.2.2 (linearizability of a program trace) *Let π be a trace of program P, and $k_1 < k_2 < ... < k_m$ a valid guess of linearization points for π. We say that* ***π is linearizable by $k_1 < k_2 < ... < k_m$*** *iff $ser(\pi, k_1 < k_2 < ... < k_m)$ is a witness for the linearizability of π by $k_1 < k_2 < ... < k_m$.*

Definition E.2.3 (induced history) *Let π be a trace of program P. The* ***history induced by π****, denoted by $H(\pi)$, is a sequence π^H of invocation and response events, corresponding to the sequence of invocation and response statements executed in π.*

I.e., let $l_1 < l_2 < ... < l_n$ be the sequence of all indices l_j such that $stmt(\pi, l_j)$ is either an inv or a res statement. Then $|\pi^H| = n$, and for all $1 \leq j \leq n$: If $stmt(\pi, l_j)$ is an inv statement then $\pi^H(j)$ is an invocation event that records the invoked operation, the argument values and the executing thread $f_\pi(l_j)$. If $stmt(\pi, l_j)$ is a res statement then $\pi^H(j)$ is a response event that records the return value and the executing thread $f_\pi(l_j)$.

Similarly, for a serial execution π^{ser}, $H(\pi^{ser})$ denotes the sequence of invocation and response events corresponding to the sequence of invocation and response statements executed in π^{ser}.

Proposition E.2.4 *Let π be a trace of program P, and $k_1 < k_2 < ... < k_m$ a valid guess of linearization points for π. Let $\pi^{ser} = ser(\pi, k_1 < k_2 < ... < k_m)$. If π^{ser} is a witness for the linearizability of π by $k_1 < k_2 < ... < k_m$, then $H(\pi^{ser})$ is a linearization of $H(\pi)$.*

Proof: Let $H = H(\pi)$ and $S = H(\pi^{ser})$. Let us extend H to H' by appending to H the following response events: For every non-terminating operation execution $i_1 < i_2 < \ldots < i_n$ in π for which there exists j such that $k_j \in \{i_l\}_{l=1}^n$, append to H the j^{th} response event of S. By Def. E.2.1, $complete(H')$ is equivalent to S. Furthermore, by Def. E.1.8, S is a legal sequential history and $<_H \subseteq <_S$. Thus, by Def. E.1.1, S is a linearization of H. $\square$

E.3 Correctness Theorem

We assume we are given a function $phase^c$ that specifies the user's choice of *fixed linearization points* for program P. This function should define a unique valid guess of linearization points for every possible trace of P, and the goal of the analysis is to verify that every trace is linearizable by this guess.

The guess for trace π, denoted by $lp(\pi)$, is $k_1 < k_2 < ... < k_m$ where $\{k_i\}_{i=1}^m = \{j \mid phase^c(f_\pi(j), \pi(j)) = PreLin \wedge phase^c(f_\pi(j), \pi(j+1)) = PostLin\}$.

The initial state of the correlating semantics is $\sigma_C^0 = \langle (\sigma_S^c)^0, (\sigma_S^r)^0, \phi_0, la_0, ret_0 \rangle$ where ϕ_0 correlates corresponding dummy nodes of $(\sigma_S^c)^0$ and $(\sigma_S^r)^0$, if any exist (see App. A.3), and la_0 and ret_0 are the empty maps.

Note that we prove the soundness of the analysis under the following assumptions: every candidate operation and every reference operation allocates, at most, a single object; the candidate operation performs the allocation before its corresponding reference operation; and the objects allocated by a candidate operation and its corresponding reference operation contain equal data items. (In all of our benchmarks, the allocated object contains a data item given as an argument value to the operation. Since corresponding candidate and reference operations are invoked with the same argument values, the data items of their allocated objects are equal.) These assumptions imply that nodes correlated by the correlating semantics have equal data items. (The dummy nodes that are correlated in the initial state σ_C^0 should also have equal data items.)

Fig. C.2 and Fig. C.3 define the transition rules of the correlating semantics. The correlating semantics essentially runs in its first two components the combined traces corresponding to the traces of P and to the fixed linearization points, while maintaining additional information in the ϕ, la, and ret components. The execution of a candidate response statement in a correlated state σ_C is allowed (by the [*finish*] transition rule) only if its return value matches that of the corresponding reference operation. If the two return values don't match, then a transition from σ_C to the *wrong* state is allowed (by the [*wrong*] transition rule). Recall that, by our definitions, corresponding return values *match* if they both have the same constant value or they are taken from the data fields of correlated nodes.

Definition E.3.1 *Let π be a trace of program P. Let $\pi^{comb} = comb(\pi, lp(\pi))$. We say that the correlating semantics* ***derives*** *π^{comb} if there exists a trace π^{corr} of the correlating semantics such that π^{comb} is the projection of π^{corr} onto its first two components. I.e.,*

- $|\pi^{corr}| = |\pi^{comb}|$, *and*
- *for all* $1 \leq i \leq |\pi^{corr}|$: *let* $\pi^{comb}(i) = \langle (\sigma_S^c)_i, (\sigma_S^r)_i \rangle$, *then* $\pi^{corr}(i) = \langle (\sigma_S^c)_i, (\sigma_S^r)_i, \phi_i, la_i, ret_i \rangle$ *for some* ϕ_i, la_i, *and* ret_i.

In this case, we say that π^{comb} *is derived by* π^{corr}.

Definition E.3.2 *Let* π *be a trace of program* P. *Let* $\pi^{comb} = comb(\pi, lp(\pi))$. *We say that the correlating semantics* ***derives wrong*** *for* π^{comb} *if there exists* $1 \leq l < |\pi^{comb}|$ *such that*

- $\pi^{comb}(1, \ldots, l)$ *is derived by a trace* π^{corr} *of the correlating semantics, and*
- $\pi^{corr}(l) \Rightarrow$ *wrong by applying the* [wrong] *transition rule.*

Lemma E.3.3 *Let* π *be a trace of program* P. *Let* $\pi^{comb} = comb(\pi, lp(\pi))$. *If* π *is linearizable by* $lp(\pi)$, *then the correlating semantics either derives* π^{comb} *or derives wrong for* π^{comb}.

Sketch of Proof: Let $\pi^{comb} = \pi_1^{comb} \cdot \pi_2^{comb} \cdot \ldots \cdot \pi_m^{comb} \cdot \pi_{m+1,c}^{comb}$ (we use the notations of Def. E.1.9).

Suppose first that we ignore the [*wrong*] transition rule of the correlating semantics and the condition $\rho_t^{c\prime}(r) = m_\phi^{-1}(ret(t))$ of the [*finish*] transition rule. Then we can construct a trace π^{corr} of the correlating semantics that derives π^{comb}, as follows: Let i be any index such that $1 \leq i < |\pi^{comb}|$.

- If both $\pi^{comb}(i)$ and $\pi^{comb}(i+1)$ are in $\pi_{j,c}^{comb}$ for some $1 \leq j \leq m+1$, or if $\pi^{comb}(i)$ is in $\pi_{j,r}^{comb}$ and $\pi^{comb}(i+1)$ is in $\pi_{(j+1),c}^{comb}$ for some $1 \leq j \leq m$, then the transition $\pi^{corr}(i) \Rightarrow \pi^{corr}(i+1)$ is enabled by applying either the [*candidate*] or the [*finish*] rule. (Note that, by definition, the condition $\forall t' \in \mathcal{T} : isOutside^r(t', \sigma_S^r)$ holds for $\sigma_S^r = \pi_1(1)$ and for $\sigma_S^r = \pi_j(|\pi_j|)$ $(1 \leq j \leq m)$.)

- If $\pi^{comb}(i)$ is the last state of $\pi^{comb}_{j,c}$ and $\pi^{comb}(i+1)$ is the first state of $\pi^{comb}_{j,r}$ for some $1 \leq j \leq m$, then the transition $\pi^{corr}(i) \Rightarrow \pi^{corr}(i+1)$ is enabled by applying the [*lin-point*] rule.

- If both $\pi^{comb}(i)$ and $\pi^{comb}(i+1)$ are in $\pi^{comb}_{j,r}$ for some $1 \leq j \leq m+1$, then the transition $\pi^{corr}(i) \Rightarrow \pi^{corr}(i+1)$ is enabled by applying the [*reference*] rule.

Now consider the [*wrong*] rule and the condition $\rho^{c\prime}_t(r) = m^{-1}_\phi(ret(t))$ of the [*finish*] rule. Suppose there exists a transition $\pi^{corr}(i) \Rightarrow \pi^{corr}(i+1)$ where we applied the [*finish*] rule in the construction of π^{corr} above, and the condition does not hold. Let l be the minimum such i. Since the conditions of [*finish*] and [*wrong*] are complementary, $\pi^{corr}(l) \Rightarrow wrong$ by applying the [*wrong*] rule, and therefore the correlating semantics derives *wrong* for π^{comb}. □

Theorem E.3.4 (Correctness of the correlating semantics)
If $(\neg(\sigma^0_C \Rightarrow^* wrong))$ *holds, then every trace* π *of the candidate multi-threaded program* P *is linearizable by* $lp(\pi)$.

Sketch of Proof: We'll show that if there exists a trace π of the candidate multi-threaded program P that is not linearizable by $lp(\pi)$ then $\sigma^0_C \Rightarrow^* wrong$.

Suppose such a trace exists, and let π be the shortest such path. Let $\pi' = \pi(1, \ldots, |\pi| - 1)$. By our choice of π and by Def. E.2.1, $stmt(\pi, |\pi| - 1)$ is a response statement and π' is linearizable by $lp(\pi')$. Furthermore, $lp(\pi') = lp(\pi)$, $ser(\pi', lp(\pi')) = ser(\pi, lp(\pi))$, and $\pi^{comb\prime} = \pi^{comb}(1, \ldots, |\pi^{comb}| - 1)$ where $\pi^{comb} = comb(\pi, lp(\pi))$ and $\pi^{comb\prime} = comb(\pi', lp(\pi'))$.

By Lem. E.3.3, there are two possible cases:

1. $\pi^{comb\prime}$ is derived by a trace $\pi^{corr\prime}$ of the correlating semantics. Let $lp(\pi) = (k_1 < k_2 < ... < k_m)$ and $ser(\pi, lp(\pi)) = \pi_1 \cdot \pi_2 \cdot \ldots \pi_m$ (we use the notations of Def. E.1.8). Let $i_1 < i_2 < \ldots < i_n$ be the terminating operation execution in π such that $i_n = |\pi| - 1$. Let j be the unique

index such that $k_j \in \{i_l\}_{l=1}^{n}$. Let $t = f_\pi(i_n)$ and $proc = pr(\pi, i_n)$. By our choice of π and by Def. E.2.1, the return value of the response statement executed in the transition $\pi(i_n) \overset{t}{\rightsquigarrow} \pi(i_n+1)$ is different from the return value of the response statement executed in the transition $\pi_j(|\pi_j|-1) \overset{proc,t}{\rightsquigarrow} \pi_j(|\pi_j|)$. In particular, the two return values cannot be the data values of correlated nodes.

The final state of $\pi^{corr\prime}$ is $\pi^{corr\prime}(|\pi^{corr\prime}|) = \langle \pi(i_n),\ \pi_m(|\pi_m|),\ \phi,\ la,\ ret \rangle$ where $ret(t)$ records the return value of $\pi_j(|\pi_j|-1) \overset{proc,t}{\rightsquigarrow} \pi_j(|\pi_j|)$. It follows that $\pi^{corr\prime}(|\pi^{corr\prime}|) \Rightarrow wrong$ by applying the $[wrong]$ rule for thread t. Therefore, $\sigma_C^0 \Rightarrow^* wrong$.

2. The correlating semantics derives $wrong$ for $\pi^{comb\prime}$. Therefore, $\sigma_C^0 \Rightarrow^* wrong$.

□